Online TA

Master Project

Datalogisk institut, Copenhagen University (DIKU)

Oleksandr Shturmov
oleks@oleks.info

May 21, 2014.

Contents

Chapter 1

Preface

I have at times went into extraordinary detail on some low-level matters of the Linux kernel. These details are provided as reference to the reader, as I know I wish I had the details at hand during the project.

1.1 Audience

The reader is assumed to be familiar with the concept of a university, i.e. an institution of higher education and research, aimed at educating scholars and professionals, granting them degrees, signifying their accomplishments.

The reader is assumed to be familiar with Computer Science, i.e. the study of computable processes and structures, with the aid of of computers. Preferably, the reader should hold a Computer Science degree, be, or have been enrolled in a Computer Science university programme.

1.2 Dictionary and Grammar

Unless otherwise stated, the reader

1.3 Legal Disclaimer

This report references certain legal documents, including Danish laws and university curricula. As the author is not trained in law, there is no claim as to the legal soundness of the claims and references made in this report.

A solid attempt has been made at retaining the formulation of the referenced material, and referencing the most current legal documents, unless this was hindered by other legal references.

For instance, the shared section of the BSc and MSc curricula for study programmes at the Faculty of Science, University of Copenhagen [Curricula (2013)] is based on Ministerial Order no. 819 as of June 29, 2010 [BEK 814]. This document is outdated and has been updated twice, most recently by Ministerial Order no. 1520 as of December 16, 2013 [BEK 1520]. In this particular

case, it is the faculty curricula that was been deemed to mandate the relevant law to reference.

1.4 References

When looking for referenced material, material published in relevant scientific journals, yet accessible to the general public, was preferred. To ensure long-lasting access to publicly available, web-based resources, they were archived using WebCite®.

1.5 About the Author

This report is written in third person out of aesthetic considerations, with the exception of this section.

When referring to "in practice", I refer to my own practice as a teaching assistant in various courses at DIKU, or the perceivable practice of teachers and teaching assistants around me.

Chapter 2

Introduction

> *In any teaching of the application of computers it is essential to have the students do practical programming problems and to grade their results. Such grading should consider both the formal correctness and the performance of the programs and tends to become difficult and time consuming as soon as the teaching is beyond the most elementary level. The possibility of using the computer to help in this task therefore soon suggests itself.*
>
> — PETER NAUR, *BIT 4* (1964)

The above quote gives about as concise an introduction to the problem at hand as we can hope to give. Being published in 1964, it also indicates that we are about to embark upon a fairly long-standing problem — we do not intend to solve it here. Instead, we take a few first steps, enabling future work.

In this chapter, we make a few grounding definitions, discuss what programming education looks like today, and discuss some of the reasons why a solution remains elusive. If you feel confident about a section heading in this chapter, feel safe to skip the section, but consider returning to it if in doubt about the terminology in a subsequent chapter.

2.1 Assessment in Education

Assessment, or evaluation, of people is concerned with obtaining information about their knowledge, attitudes, or skills. Assessment in education is usually concerned with obtaining information about students and their learning[1] [Ramsden (1992), Pishghadam et al. (2014)]. This is done with the intent of providing feedback or certification, performing selection or comparison, improving learning processes, etc. [Bradfoot & Black (2004)].

2.1.1 Categorising Assessment

There are two principal categories of assessment: formative and summative. The definition of each category varies somewhat in educational research [Bloom

[1] Sometimes, assessment in education is concerned with evaluating teachers and their teaching — we will not be concerned with this form of assessment here.

et al. (1971), Sadler (1989), Harlen & James (1997)], and their mutual compatibility is questionable [Butler (1988)]. Our intent is not to advise on the matter, but to aid in performing an assessment, regardless of the flavour.

Let us adopt a primitive distinction, which still supports the purposes of our further analysis:

Formative
: A student's strengths and weaknesses are documented in free form. Formative assessments are qualitative and non-standardised: they are aimed at measuring the quality of a student's learning, rather than whether they live up to particular criteria.

Summative
: A student is ranked on some well-defined scale, at some well-defined intervals, based on some well-defined criteria. Summative assessments are often compoundable and comparable. They may allow to deduce holistic summative assessments of students or groups, quantitatively measure student progress, etc.

Formative assessment necessitates the ability to perform personalised assessments, whereas summative assessment demands the ability to specify standards and perform standardised assessments.

There are other forms of assessment: diagnostic assessment, self-assessment, peer-assessment, etc. [Bull & McKenna (2004), Topping (1998)]. These forms of assessment vary along formative/summative dimensions, but primarily differ in terms of when, by whom, and of whom the assessment is made.

2.1.2 Feedback

Feedback is information about the difference between the reference level and the actual level of some parameter which is used to remedy the difference in some way [Ramaprasad (1989)].

Feedback is an important bi-product of assessment in education [Black & William (1998)]. Ideally, feedback informs the student of the quality of their work, outlines key errors, provides corrective guidance, and encourages further student learning. To be so, it is important that feedback is understandable, timely, and acted upon by students [Gibbs & Simpson (2004)].

These requirements are an active area of research in education. One aiding approach is to use computer-assisted assessment.

2.1.3 Computer-Assisted Assessment

Computer-assisted assessment is the form of assessment performed with the assistance of computers [Conole & Warburton (2005)]. The benefit of using computers is ideally, fast, interactive, consistent, and unbiased assessment [Ala-Mutka (2005)]. The requirement is that the perceived student performance can be encoded in some useful digital format.

This requirement however, has proven elusive. Free form performances, such as essays or oral presentations, are still hard to assess automatically [Valenti

et al. (2003)]. On the other hand, it is questionable in how far easily assessable performances, such as, multiple-choice questionnaires, are appropriate for assessment in higher education [Conole & Warburton (2005)].

We conjecture, that in how far computers can assist in assessment, depends on how "rigorous" the student performance can be expected to be. We formalise this notion below.

2.1.4 Assignment

An assignment is a request for someone to perform a particular job. An assignment in education is a request for a student to make a performance, and often, to provide a record thereof. One purpose of an assignment is to provide a basis for an assessment. The request therefore, often includes a specification of what the assessment will be based on, and in what time frame the assignment should be completed in order to be assessed.

2.1.5 Submission

A submission is a record of student performance, submitted for the purposes of assessment. A digital submission is a digital encoding of such a record. Digital submissions are amenable to assessment with the assistance of computers.

2.1.6 Rigor

We say that the more features of interest can be extrapolated from a data structure using efficient algorithms, the more "rigorous" the data structure. Rigorousness therefore, depends on the features that we are interested in.

A data source is rigorous if the data it delivers is rigorously structured, and the same suite of algorithms can be used for all the data it delivers. We conjecture that in how far computers can assist in assessment depends on how rigorous students, the source of submissions, can be expected to be.

The use of computers for structuring submissions can sometimes provide for high rigor. For instance, in a multiple-choice test, a computer may present the student with the questions and options. The student may then respond to the computer using toggles, and have the computer encode the choices in a tableau. An assessment then constitutes merely comparing against a reference tableau — something computers are notoriously good at.

If instead, the student is asked to write an essay in a natural language, today's computers can assist with little more than dictionaries, thesauri, grammar, and mark up. Although this provides for some rigor, natural languages are at best, somewhat rigorous. The extent of this "somewhat" is the subject matter of research in natural language processing and automated essay assessment [Valenti et al. (2003)]. Natural languages however, are much more expressive, allowing for much richer assessment [Conole & Warburton (2005)].

Beyond where computers can help, it is indeed the question of how rigorous one can expect the students to be. In some disciplines, such as programming, high rigor can often be expected in submissions. We explore this notion in the following sections.

2.2 Courses

A course is a unit of education imparted in a series of learning activities. A student is someone who is enrolled for a course for the purposes of learning. A teacher is someone who is enrolled for a course for the purposes of teaching. Other roles are discussed in a subsequent section.

A course has some predefined knowledge or skills that the students are expected to acquire by the end of the course. It is the teachers that impart this knowledge or skills onto the students, in hope that they retain it.

The student performance is typically summatively assessed at the end of a course — at least, on a pass/fail/neither basis. A student passes a course, if the student has shown to have acquired the said knowledge or skills, and fails otherwise. A student may neither pass nor fail in various extraordinary cases, such as the student dropping out of a course before a final assessment.

This superimposes that submissions to particular assignments must be summatively assessed — at least, on a pass/fail/neither basis. We say that a student "passes" an assignment, if some submission is assessed as "passed".

Formative assessments of submissions are typically conducted throughout the course to facilitate and courage student learning, and sometimes also at the end, to facilitate and encourage future learning.

In a subsequent course evaluation, students may assess how well the teachers performed in their teaching. We will not be concerned with this here.

2.3 Assessment in Computer Science

Computer Science is the study of computable structures and processes. Computer Science professionals are (among other things) expected to be eloquent in the theory and practice of computer programming [CS Curricula 2013].

To this end, practical work is a popular basis for assessment in Computer Science [Carter et al. (2003)]. Practical work is concerned with the composition of programs to be executed by a computer, solving a particular problem.

To be executable by computers, computer programs are often written in highly rigorous languages, and so are amenable to assessment with the assistance of computers. The assessment of computer programs is a wide area of research and industry, known as software verification or quality assurance.

The use of computers for structuring a submission for a programming assignment can provide for some rigor. Students can be, and often are, expected to acquire skills in using various programming tools on their own account, or with some facilitation from their teachers.

2.3.1 Programming Languages

Assignments in Computer Science will often ask students to write computer programs in one of a range of different programming languages. Programming languages come and go, and no language has emerged as the predominant one for teaching Computer Science. It is most useful, therefore, to facilitate assessment of programs written in any conceivable programming language.

Modern programs always run within the context of an operating system. One way to stay programming language agnostic, is to facilitate assessment at the operating system level. This requires looking into how student programs, and various analyses thereof, can be run in safe and fair environments, and how the various elements of an assessment engine can communicate via the operating system in a safe and secure way.

2.3.2 Style Requirements

In educating programming, we are only partially concerned with teaching students to build executable computer programs solving particular problems. Many programs are written in collaboration with others, and people other than the authors often end up testing and maintaining the source code.

It is important that students learn to write readable, testable, and maintainable source code. This means writing source code so that it can be quickly grasped by others, using the right abstractions, and using the abstractions right. This includes e.g. checking adherence with a style guide.

2.3.3 Permissions and Restrictions

Assignments in Computer Science will often also ask students to write programs that operate within environments with particular permissions and restrictions. We should facilitate such permissions and ensure that the restrictions are adhered to. Both enable certain types of assessment. For instance, we may wish to permit that students can write to a particular file, but restrict how many I/O operations they may perform in total.

2.3.4 A Reference Course

A good reference course, is a course in operating systems programming in C.

C is ubiquitous on modern computer architectures, with the folklore that the first thing you should write for your new processor is a C compiler. We conjecture, that anything written for execution on a modern computer, with sufficient effort, can also be written in C. Especially because we can write assembly in C. C is also ubiquitous on modern operating systems. Perhaps the most basic API that a modern operating system offers is a C API. If not, we can again resort to writing assembly in C.

Operating systems courses typically also involve a wide range of necessary permissions and restrictions. Especially if student programs are not written for some idealised, virtual environment (e.g. Buenos), but are intended for running in the context of a real operating system.

Such programs may run into faults, make obscure system calls, access devices, perform I/O, not to mention run high on memory use and CPU time. We conjecture, that if we can facilitate safe and fair environments for a course in operations systems programming in C, we can facilitate safe and fair environments for most courses in Computer Science.

We will however, make a few simplifying assumptions as we seek to meet certain safety, fairness, and security requirements. For instance, we will as-

sume that students do not need access to networking facilities, or access to more than a handful character devices.

2.4 Roles in Educational Assessment

In the previous sections we introduced the concepts of a student and teacher, enrolled for a course. In this section we expand on the roles of course enrollees.

2.4.1 Teachers

Teachers are enrolled for a course for the purposes of teaching. Teaching is the expediting of learning. Students learn on their own, but teachers facilitate learning [Skinner (1965)]. The means of facilitation however, vary greatly throughout the discipline [Ramsden (1992), Kember (1997)]. Most would unite the role of teaching with information delivery and assessment.

A non-empty set of teachers is always held responsible for a course. They are responsible for ensuring that the students acquire the knowledge or skills defined for the course. Other teachers may be involved in the course, aiding the course responsible teachers.

In hope of students' learning, the teachers devise means of delivering informative content, and assessing in how far students have acquired the said knowledge or skills. Various techniques in both information delivery and assessment are used to facilitate and encourage learning. We are not so concerned with information delivery, as with assessment. Never-the-less, assignments still have to be delivered to students.

Since teachers facilitate student learning and also their final assessment, teachers exert great authority over students. Teachers decide whether a student passes or fails a course, what grade they get, and how tough the course is.

Teaching assistants

Teaching assistants assist in teaching responsibilities. They are teaching subordinates of teachers. They exert some authority over students, but are often limited in their authority when it comes to important summative assessments. The result is that teaching assistants perform much of the formative assessment, and provide guiding remarks either for the purposes of feedback or to ease important summative assessments for the teachers.

Teaching assistants come about as a scaling mechanism. Once the number of students enrolled for a course exceeds a certain multiple of teachers, certain means of information delivery and assessment are simply infeasible for the teachers. Instead of hiring more teachers, the strategy is often to rely on some methods of information delivery and assessment that work in large numbers, and rely on teaching assistants for the rest. Teaching assistants are cheaper, often less qualified, staff who assist in teaching responsibilities.

We refer to teachers and teaching assistants collectively as teaching staff, or simply, staff.

Trust

Under the authority of the course responsible, the teachers and teaching assistants can be trusted to make fairly good assessments of student performance.

In various disciplines, especially in Computer Science, assessment involves some fairly mechanical processes. Teachers and teaching assistants quickly develop a desire to get a computer to do what is otherwise fairly laborious work. To this end they write programs and set up processes that analyse student submissions.

Teachers and teaching assistants can in general be trusted to write programs that often perform the analyses correctly, but bugs may lure. It is desirable that automated assessments can be validated and reverted by the teachers and teaching assistants. It is also important that their programming mistakes don't get in the way of student learning, by e.g. consuming all system resources.

2.4.2 Students

Students are enrolled for a course for the purposes of learning — the acquisition of knowledge or skills. Learning is a qualitative change of an individual's view of the world [Ramsden (1992)].

Trust

Ideally, students engage in learning in hope of being somehow enlightened or trained for solving particular problems. Realistically, student motivation varies greatly, and may even change throughout a course. Some are motivated by the mere idea of a good final assessment, e.g. to impress an perspective employer, friends, or family.

Unlike teachers, students have few direct responsibilities. In how far they actually acquire various knowledge or skills, often only bears consequences much later in life. Students may therefore be interested in the attainment of deceptive assessments, claiming that they have acquired various knowledge or skills, when in reality they were never assessed to have done so. This involves hacking the assessment process with the intent of cheating, or faking statements of accomplishment.

Students also can not be trusted to always be nice to their peers. Due to various motivations they may wish to abuse or tamper with the assessments of other students. Without further ethical consideration, it is perhaps best to let assessments be a personal matter.

More often however, students just make mistakes, sometimes big mistakes. They cannot be trusted to write good programs in their first or any subsequent attempt. One of the benefits of online automated assessment, can be that students can fail fast and fail often. This may facilitate learning.

2.4.3 External Examiners

External examiners facilitate the quality assurance of assessment. An external examiner's participation in an assessment may vary from mere observation to

avid participation. An external examiner therefore may need varying access to the elements of a course and assessments.

2.4.4 General Public

The general public includes those who are ultimately interested in the quality of education and the quality of assessment therein. This includes both perspective students, future employers, the politically concious, etc. The intent being to see if the education lives up to social expectations, demands of the labour market, political promises, etc.

Privacy and anonymity is often a matter public concern. If access is granted to the general public, it should only identify those who may reasonably be held responsible for an possible shortcomings. Also, issues of copyright have to be taken into account.

As the general public would assess education, and not students, students should not be personally identifiable by the general public. In how far teachers and teaching assistants may reasonably be held responsible by the general public for the possible shortcomings, may be a matter of policy of the overarching authority (e.g. a university).

As students typically own the content they produce, individual student work or commentary should not be made available to the general public. In how far teachers and teaching assistants own the content they produce, may again be a matter of policy of the overarching authority.

Chapter 3

Analysis

Students make submissions in response to assignments. We would like to assess their work with the assistance of computers.

We can rely on students to attempt to deliver their submissions in a digital format, over an insecure network, e.g. the Internet. The network is insecure in the following sense: it cannot guarantee that a particular submission (a) comes from a particular student, nor (b) that it has not been tampered with by an adversary.

In assessing programming assignments, we distinguish between two types of assessment. Static assessment is the analysis of a submission without executing student programs. Dynamic assessment is the analysis of the runtime behaviour of student programs. A static assessment often enables a subsequent dynamic assessment, and sometimes runs in lockstep with it.

3.1 Static Analyses

We may wish for a digital submission to adhere to a particular format, but we cannot rely on students to meet such requirements in general. One basic use of computer-assisted assessment is checking whether a submission meets certain formatting requirements, before any further assessment.

For a programming assignment, we may wish for a submission to consist of a computer program and a report — we'll use this as a running example.

It is fairly straight-forward to enforce the requirement that a digital submission be a non-empty set of digital files, distinguished by their file names. Similarly, we can require files with particular names to be submitted, stating a particular file type. For instance, two text files, named `main.c` and `report.txt`. It is also fairly straight-forward to enforce a limit on the sizes of the files in a submission. This has some security benefits.

We may further enforce the requirement that particular files look like they are written in particular languages. For instance, that the computer program is written in C, and the report is written in English. We say "look like" because neither language in this example has a formal definition, and so cannot be guaranteed to be recognisable by a computer, but both are in frequent use.

For a lot of programming languages — we're in luck — the job of a parser, as part of e.g. a compiler or assembler, is indeed to recognise, whether a sequence of bits can be interpreted as a statement in that language[1].

With natural languages, being less formal in general, we're in a bit less luck. We can often analyse various features of a report probabilistically, with fairly high confidence. Notably, such probabilistic techniques give only limited corrective guidance. We conjecture that reports will require subsequent assessment by the teaching staff much more often than computer programs.

Checking that a submission meets such requirements is usually a matter of static analysis. C++ Templates aside [Veldhuizen (2003)], checking that a submission meets programming language requirements should not result in the execution of programs written by students in a Turing-complete language.

Putting a C program through a C compiler, we get an executable out. This enables a subsequent dynamic analysis. For other languages, we may perhaps only have a (lazy) interpreter, where the syntax of a part of a program is (only) checked immediately before it is executed. This would require checking a valid syntax requirement in lockstep with a dynamic analysis.

We will often also want to check that the program is readable, testable, and maintainable by others, that it uses particular programming abstractions, and uses them right. This superimplies that we want submissions to include the source code of the programs. The above requirements can then be checked by checking that the source code adhere to comprehensive, assignment-specific, style guides.

This amounts to a range of static analyses that we may want to perform, some of them in lockstep with dynamic analyses.

3.2 Dynamic Analyses

We are not only concerned with students submitting good-looking source code, but also that their programs solve the problem at hand. Sometimes, this can be answered by a static analysis. More often however, we must resort to executing the student programs and analyzing their runtime behaviour.

Such programs often require some input data, and produce some output data — they may even be interactive[2]. We need to generate data for our student programs and validate that the data they output is correct. For a particular assignment, we make the following observations:

(a) in how far an output is correct, may depend on the input;

(b) there may be more than one correct output for the same input.

Other than produce wrong results, student programs may misbehave in a myriad of different ways, perhaps even intentionally. If let to their own devices, student programs may never terminate, abuse memory, leak memory,

[1] The question of whether a sequence of bytes can be interpreted as e.g. a C program, is often reduced to whether or not it is recognisable by a particular C compiler.

[2] For simplicity, we currently ignore assignments that require building a graphical user interface. This still leaves rich interactiveness capabilities.

fiddle with devices, make obscure system calls, and generally fail in unpredictable ways.

Of course, this also applies to static analyses. A static analysis can be thought of as an idealised computer, whose instructions are fed by the input to the analysis. The designer of a static analysis may not anticipate all the possible failure scenarios, or even let the language it accepts be Turing-complete.

Such intentionally or unintentionally misbehaving analysis of submissions may interfere with other analyses on the system, causing faulty assessments, or even discarding assessments.

We must protect against such abuse for both the static and dynamic analyses. We conjecture that the latter will be (unintentionally) abused more often than the former. We conjecture that good feedback on either abuse, but especially the dynamic assessment, can be an effective learning tool. For instance, a tight bound on time and space may train students to write more efficient programs.

As we protect against such abuse, we must monitor the execution of student programs. The data gathered from this monitoring may also be useful for further feedback to the student.

3.3 Feedback

In case of failure, the feedback from the static and dynamic analyses may be generally incomprehensible to a human being [Lerner et al. (2007)], let alone a student learning to program [McCauley et al. (2008)].

In general, feedback is an important pedagogical tool. We would like to support processing the feedback, before it is delivered to the student. Such a processor may additionally be parametrised by the original submission and some student data to support e.g. individualized learning.

This processing may include compounding feedback from the static and dynamic analyses to form a final feedback, providing hints to solve problems (instead of presenting error messages), or just pass or fail the submission without further note. We do not consider feedback processing beyond the notion that it is a parametrised process defined on a per-assignment basis. Providing good feedback in response to programming assignments is beyond the scope of this work.

Feedback processing may include having a member of the teaching staff look over the feedback to make sure that it is correct, and often, add to it. For instance, the system may (a) validate that the student submission contains a computer program and a report, (b) validate that it is written in the intended language, (c) validate that it uses proper style, and expected programming abstractions, (d) validate that it produces valid results and runs within the given resource bounds. This leaves it to the teaching staff to validate that the report properly reports about the written program.

Human involvement does not need to happen before a student is informed of the original (automated) feedback. For instance, a common scenario might be that the student makes a sequence of submissions, in attempt to pass the automated testing. Once a submission passes, they can move on to a new as-

signment, while they wait for more extensive feedback on their submission. The teaching staff then, only look at the submission that passed the automated testing. Retaining old submissions may be useful to assessment in general.

3.4 Course Content

The feedback aside, the teaching staff would like to deliver the assignments themselves and various learning material to students. We conjecture that gathering all of these elements into one system provides a better interface between the students and the teaching staff.

Having a single system for all this content opens up authenticity and authorisation concerns: we would like to make sure that all users of the system corresponds to a particular enrollee on the course (authentication), and that their user permissions correspond to their enrollee roles (authorisation).

3.5 Other Notions

In the past we've used notions such as "safety", "fairness", and "security". In this section, we formalise these notions wrt. our system.

In general we seek that the running of other processes on the system does not interfere with the assessment of a submission. Other processes may be other assessments, or other parts of the system. This subdivides further into a set of safety, fairness and security requirements.

For simplicity, we will assume that the system retains all course data, until all of it is securely wiped by a system administrator. It follows that some of the data may loose integrity or become vulnerable over time.

3.5.1 Safety

Safety requirements are concerned with ensuring the integrity of assessments over time: A process p should not interfere with an assessment a, where $a \neq p$, in the sense that the feedback generated from assessment a, is the same, regardless of whether process p is ever executed.

3.5.2 Fairness

Fairness requirements are concerned with ensuring that all assessments get a fair share of system resources: A process p should not interfere with an assessment a, where $a \neq p$, in the sense that the assessment a gets allocated the same system resources, and should run in roughly the same wall clock time, regardless of whether process p is ever executed.

If resources and wall-clock time cannot be guaranteed, an assessment should be queued on a first-come-first-serve basis, provided that a student can have at most one submission assessed at a time.

3.5.3 Security

Security requirements are concerned with ensuring that no assessment is made vulnerable over time. An assessment is made vulnerable if it reveals data to unauthorised users.

In , we discussed various enrollee roles for a system. Below, we whitelist permissions for all these roles.

Teaching Staff
: Can see all versions of and create new (or new versions of) learning content and assignments. Can see all student submissions as well as processed and unprocessed feedback. Can create new versions of the processed feedback, adding to a previous version or discarding some of it.

Students
: Can see all versions of the learning content and assignments. Can create new personal or group (done in a group with other students) submissions in response to assignments. Can see all past personal or group submissions. Can see all versions of the processed feedback on the above submissions.

General Public
: Can see all versions of the learning content and assignments.

Chapter 4

Sandboxing untrusted code

Going all the way back to early time-sharing systems, we systems people regarded the users, and any code they wrote, as the mortal enemies of us and each other. We were like the police force in a violent slum.

— ROGER NEEDHAM, IEEE Symposium on Security and Privacy (1999)

Students submit digital files in response to assignments. Some of these files may specify executable computer programs. The automatic evaluation of student submissions constitutes the static and dynamic evaluation of such files. Static evaluation constitutes executing computer programs specified by the teaching staff, which read and analyze student files. In addition, dynamic evaluation includes executing the programs submitted by students.

The student programs may misbehave in a myriad of different ways. The programs of the teaching staff, although more trustworthy, may also misbehave. If nothing else, they may undermine the misbehaviour of students. The intent of this chapter is to discuss the means in which we can mitigate for such misbehaviour for all parties, and ensure fair service.

In the first section we provide a high-level overview of the technologies that can be used sandboxing. Here we come to the conclusion that operating-system level virtualization is a best candidate option. The remainder of the chapter deals with basic principles of virtualizing and limiting system resources in [Linux kernel (v3.14.2)], henceforth the Linux kernel.

4.1 Technology Overview

A program is executed within a program execution environment. A sandboxed execution environment ensures the non-interference of the program with other programs being executed on the host.

There are two general approaches to providing sandboxed program execution environments: sandboxing the operating system, or sandboxing within an operating system. The first two subsections discuss the former, the last subsection discusses the latter.

4.1.1 Dedicated Servers

We can provide an off-the-shelf operating system sandbox using a dedicated server for every program. This however, relinquishes remote control of the execution environment, and may demand physical access to the machine in case of failure. This is impractical. Also, this is expensive since most computer systems today are intended as time-sharing systems.

4.1.2 Virtual Machines

The next option is to provide an operating system by means of hardware or software virtualization. This retains remote control of the execution environment. However, it imposes huge costs on every execution. An entire operating system has to boot up before testing can commence.

Alternatively a pool of virtual machines could be kept online, pulling tasks from a task queue. Such a set up does not always fail fast, again because an entire operating system may have to be rebooted in case of failure. Combined with empirical evidence that student programs fail often, this is impractical.

Such high-level virtualization also makes the execution environment hard to monitor. The overhead of the operating system may dilute the true costs inherent in executing various programs. For similar reasons, fine-grained resource limits are often hard to enforce. All this is desirable for the purposes of evaluating our programs and tuning our sandboxes.

4.1.3 Operating system-level virtualization

Operating system-level virtualization alleviates the need for a separate kernel for sandboxing program execution environments.

Time-sharing systems have for a long time provided for multiple simultaneous user space instances on top of a single kernel. Combined with file-system user permissions and user groups, these provided for the very first sandboxing capabilities.

Recent developments in modern operating systems have facilitated more fine-grained sandboxing by virtualizing underlying system resources. Such a virtualized user space instance is typically called a “jail” or a “container”.

The pitfall of operating system-level virtualization in general, is that we become more vulnerable to vulnerabilities in the kernel. If a contained program can utilize a kernel vulnerability, the whole system is under threat.

FreeBSD Jails

Linux Kernel Containment

- LXC
- libvirt-lxc
- Docker

4.2 Control Groups

Control groups (cgroups) provide a mechanism of hierarchically grouping/-partitioning tasks (see also Appendix A.1/59) and their future children [cgroups.txt]. On their own, cgroups are perhaps only useful for simple job tracking. The idea, is to have other subsystems hook into the cgroups functionality and provide for management of system resources.

The standard cgroup subsystems include subsystems to monitor and limit memory, CPU time, I/O, and device activity. Subsystems therefore are often also called "resource controllers". Many modern Linux distributions come with cgroups and many of these standard subsystems enabled. The system's `/proc/config.gz` can reveal the setup on your system [proc(5)]. If `CONFIG_CGROUPS` is enabled, you have cgroups support.

The variables related to various subsystems are explored further in the following subsections. First however, we discuss how cgroups can be accessed and manipulated from user space in general, as well as a general framework for resource management.

4.2.1 Managing cgroups

Cgroups are managed via a pseudo-filesystem: cgroups reside in memory, but can be manipulated through the virtual file system. `cgroup` is therefore an inherent file system type on systems that have the cgroups functionality enabled.

Cgroups, subsystems, and hierarchies

A cgroup is an association of a set of tasks with a set of preferences for a set of subsystems. A hierarchy is a set of cgroups arranged in a rooted tree. Every task in the system is attached to exactly one cgroup in the hierarchy. All cgroups in the hierarchy, associate their tasks with the same set of preferences — we say that a hierarchy is associated with a set of subsystems[1]. Figure 4.1/25 illustrates a couple example hierarchies.

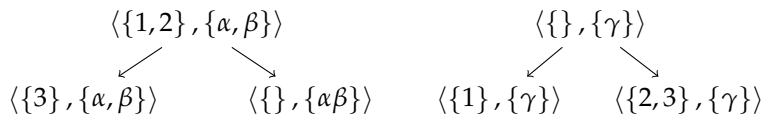

Figure 4.1: An illustration of two example cgroup hierarchies on a particular system. The set of identifiers of the tasks running on the system is $T = \{1,2,3\}$. The set of subsystems available on the system is $S = \{\alpha,\beta,\gamma,\delta\}$ (not all subsystems need be associated with a hierarchy). Every node (cgroup) has type $\mathcal{P}(T) \times \mathcal{P}(S)$, where $\mathcal{P}$ denotes the powerset.

[1] [cgroups.txt] is ambiguous wrt. whether a hierarchy can exist without being associated with at least one subsystem. For the sake of simplicity, we'll assume that it can; although cgroups without associated subsystems have few practical applications, as already discussed above.

Mounting

When mounting a cgroup file system, we create a new hierarchy. The set of subsystems to associate with the hierarchy is listed as mount options[2]:

```
$ mount -t cgroup -o cpu,cpuacct cgroup ./cgroup/cpu,cpuacct
```

This associates the `cpu` and `cpuacct` subsystems with a new hierarchy, and mounts the hiearchy under the target `./cgroup/cpu,cpuacct`, unless one of the subsystems is busy.

A subsystem is busy e.g. if it is associated with a hierarchy having tasks attached. Since a hierarchy is automatically attached to all tasks in the system, this effectively means that a subsystem may be associated with at most one hierarchy. If a hierarchy assosciated with the exact same set of subsystems already exists however, it will be reused for the new mount.

What hierarchies already exist, and what subsystems they are associated with, depends on the system at hand. The system's `/proc/mounts` can reveal how this is setup on your system [proc(5)].

Control files and child groups

After a hierarchy is successfully mounted, we see a range of files, and perhaps folders, below our target. We refer to these as control files and child groups, respecitvely.

We monitor/modify the preferences of a cgroup by monitoring/modifying the control files. We create/remove child groups by creating/removing subdirectories below our target.

With a few exceptions at the root of the hierarchy, all cgroups contain the same files, created when the cgrpup is created. Some files are common to all hierarchies, others are due to the associated subsystems. Two common files are of particular interest:

`cgroup.procs`
: Lists the set of thread group IDs in the current cgroup. Appending a thread group ID to this file moves all the threads in the thread group into this cgroup.

`tasks`
: Lists the set of thread IDs in the current cgroup. Appending a thread ID to this file moves the thread into this cgroup.

Hierarchical accounting

Hierarchical accounting is when resource accounting is child group aware. All resource usage is summed up for all tasks in the cgroup, and recursively for all child groups. Limits are then imposed on the entire hierarchy. A subsystem does not necessarily perform hierarchical accounting.

[2] Omitting the subsystem list, attempts to associate all subsystems available on the system.

4.2.2 The Resource Counter

The resource counter is a framework for managing a resource when using control groups [resource_counter.txt]. The internal data structures aside, the framework makes recommendations wrt. the control files. A couple of the recommended control files are of interest to us:

`<resource>.max_usage_in_<unit_of_measurement>`
: Reading this file, we get the maximal usage of the resource over time, in the given units. Writing to this file, resets the value to the current usage of the resource. (The data written is ignored.)

`<resource>.limit_in_<unit_of_measurement>`
: Reading this file, we get the maximal allowed usage of the resource, in the given units. Writing to this file resets the limit to the given value. A special value may indicate no limit.

These files are of interest to us as they allow us to probe the usage of a resource in a test instance and set up resource limits for students or staff.

4.2.3 memory

The memory subsystem allows us to monitor and limit the memory usage of the tasks in a cgroup [memory.txt]. This includes both user and kernel memory and swap usage. The subsystem optionally performs hierarchical accounting.

Due to the considerable overhead of memory and swap accounting, some distributions do not enable this cgroup, or merely do not enable swap accounting by default. The latter is especially misleading. If swapping is enabled, a memory limit with no swap limit has at best a hapless effect.

You can check the setup on your system by checking the options prefixed with `CONFIG_MEMCG_` in your `/proc/config.gz`. Swap accounting can be enabled using the standard kernel parameter `swapaccount=1` [kernel-parameters.txt]. Enabling the memory cgroup can be a little more distribution-specific. In a Debian kernel, this can be done using the kernel parameter `cgroup_enable=memory` [Hutchings (2011)].

The memory subsystem uses a resource counter for a couple different memory resources. The resource counter control files (see also § 4.2.2/27) are prefixed as follows:

`memory`

: The main memory counter. This includes both user and kernel memory.

`memory.memsw`

: The main memory, plus swap. Limiting this value to the same value as the main memory controller, disables swap.

`memory.kmem`

Kernel memory. All kernel memory is also accounted for by the main memory counter. It is not necessary to limit this value if swapping is disabled and there is a limit on the main memory counter (since kernel memory cannot be swapped out).

`memory.kmem.tcp`

Kernel TCP buffer memory. Although we will disallow networking in general, it might be a good idea to 0-limit this resource as an extra precaution.

The limits and usage are always measured in bytes. Setting the limit to `-1`, removes the limit on the resource.

The memory subsystem does not necessarily perform hierarchical accounting. This can enabled by writing `1` to the `memory.use_hierarchy` control file in the root cgroup.

4.2.4 cpuacct

The CPU accounting (cpuacct) subsystem allows us to monitor the CPU time usage of the tasks in a cgroup [cpuacct.txt]. The cpuacct subsystem always performs hierarchical accounting.

The cpuacct subsystem provides a couple control files of interest:

`cpuacct.usage`
Shows the total CPU time spent by the cgroup, in nanoseconds.

`cpuacct.usage_percpu`
Shows the total CPU time spent by the cgroup, for each CPU core, in nanoseconds.

`cpuacct.stat`
Shows a further division of the CPU time spent. For now, showing how much of the CPU time was spent running in user mode, and how much in kernel mode, in the `USER_HZ` time unit.

4.2.5 cpu

The cpu subsystem facilitates CPU scheduling parameters for a cgroup [sched-design-CFS.txt, sched-bwc.txt, sched-rt-group.txt]. The parameters currently facilitate control over two different schedulers in the Linux kernel:

Completely Fair Scheduler (CFS)
A proprtional share CPU scheduler. The CPU time is divided fairly among tasks depending on their priority and the share assigned to their cgroup.

Real-Time Scheduler (RT)
A real-time scheduler for real-time tasks, i.e. tasks for which it is important to meet deadlines. For real-time tasks, a particular amount of CPU time must be guaranteed over a particular period of time.

For the RT scheduler, the subsystem parameters facilitate limiting how much CPU time the real-time tasks in a cgroup may spend in total over a period of time. Enforcing such limits and meeting real-time deadlines seems like a heedful task. For simplicity, we'll disallow students from spawning real-time tasks. In a default setup, spawning real-time tasks requires privileged access, which we already do not grant to our sandboxed programs.

The CFS parameters facilitate first-and-foremost the enforcing of a lower bound on the amount of CPU time allocated to a cgroup. This is done by assigning a relative share (weight) to a cgroup. The shares are enforced, only if tasks from different cgroups are competing for CPU time. This means that if a cgroup gets no competition, it gets all the CPU time it wants.

With the advent of "cloud computing" however, it has also become relevant to facilitate enforcing upper bounds on the CPU time over a period of time [Turner et al. (2010)]. This is facilitated similarly to the RT scheduler.

We choose to let the students spend all the CPU time they want, as long as fair service is ensured for all students and staff. There is therefore only one control file in this subsystem of interest to us:

`cpu.shares`
: Show/set the relative CPU time share of a cgroup. Two cgroups having share 100, will be given equal service. If one of the groups has share 200, it gets twice as much CPU time under a fully-loaded system. This control file in the root cgroup, provides for a yard-stick for all other cgroups.

All these options do not allow us to hard limit the amount of CPU time used by a cgroup in total. To our knowledge there is no "natural" way of doing this in the Linux kernel. We must make due with limiting the wall-clock running time of an untrusted program.

4.2.6 cpuset

The cpusets subsystem allows us to assign a set of CPU cores and a set of memory nodes to a cgroup. This can be used to further partition system resources from a fairly high level.

We choose to let the system be runnable on commodity hardware. With few processor cores, and few memory nodes, this subsystem is of little use to us.

4.2.7 devices

The devices subsystem allows us to mandate access to device nodes (files) using cgroups [devices.txt]. The limits are enforced hierarchically using whitelists — a cgroup further down in the hierarchy cannot access devices to which access has not been granted further up in the hierarchy.

A whitelist entry consists of four fields: the device node type, the major and minor device node identifiers (2 fields), and an access specifier. The access specifier is a sequence of characters, where `r` signifies read access, `w` write access, and `m` device node creation.

The device node type is either `c` for character, `b` for block, or `a` for all devices. `a` is a wildcard. Using it, discards all other options, and implies full access to

all devices of all types. This is useful if you want to mandate universal access. The major and minor device identifiers is what Linux uses to uniquely identify devices. The device node type, its major and minor identifiers of can be found using `stat`.

```
$ stat --format "type:%F, major:%t, minor:%T" /dev/urandom
type:character special file, major:1, minor:9
```

The following control files facilitate device whitelist management:

`devices.allow`
: Writing an entry to this file adds an entry to the device whitelist. Reading is prohibited.

`devices.deny`
: Writing an entry to thisfile removes an entry from the device whitelist. Reading is prohibited.

`devices.list`
: Shows the device whitelist. Writing is prohibited.

The format of a whitelist entry is `<type> <major>:<minor> <spec>`, where `<major>` and `<minor>` can be `*` indicating all versions. Writing `a` to `devices.allow` or `devices.deny` is the same as writing `a *:* rwm` to the same file.

4.2.8 blkio

The Block IO subsystem allows us to monitor and mandate access to I/O operations on block devices using cgroups [blkio-controller.txt]. The monitoring parameters provide for insight into the I/O performance of a cgroup. The access mandating parameters provide for proportional and absolute limits on the number of I/O operations by a cgroup.

We choose to not provide access to block devices to students in general. This is done due to the ease of implementation of a particular security policy. Giving unmandated access to a block device, in theory, gives unmandated access to all data on that device. Protecting data on such a device is a complicated matter. Although SELinux, with it's per-inode restrictions, could presumably be used to this end, it is easier to just not give access to block devices in general.

As will be discussed later, this does not inhibit us in providing students with a general purpose read/write file system.

4.3 Namespaces

The purpose of a Linux namespace is to abstract over a system resource, and make it appear to tasks within the namespace, as though they have their own isolated instance of the global resource. Various namespace types abstract over various system resources.

Namespaces are hierarchical in the following a sense: A system boots with one global namespace for each namespace type. A task created by a task within a particular namespace, is put in the same namespace by default. A task can

be associated with a new child namespace, or some other namespace already in the hierarchy.

We say that a parent namespace is a "host", and a child namespace is a "container". A host may host many containers, and a container may have many hosts, namely all of its ancestors. Typically, we'll only talk about a host and a container in a direct child-parent relationship.

A task can be associated with a namespace using the [unshare(2)], [setns(2)], or [clone(2)] system calls. The first disassociates the process from a namespace, associating it with a new namespace. The second reassociates the process with an existing namespace. The last is the general system call for task creation, allowing to create a task, already in a new namespace.

The namespaces that a task is associated with are identified in the [proc(5)] pseudo file system. The general pattern of the file names is `/proc/[pid]/ns/[nstype]`, where `[pid]` is the thread group identifier of the task (see also Appendix A.1/59), and `[nstype]` is one of a range of supported namespace types.

We discuss some of these types in the following sections. Support for more types may come in the future, as containers demand more resource isolation. Furthermore, not all of these types are necessarily enabled on your system. The user namespace is frequently omitted by many distributions as it opens up a large part of the kernel, previously not available to the non-privileged user. Some believe it requires a lot more testing before being enabled by default.

To our knowledge, enabling a namespace type requires compiling your own kernel. You can check which namespaces are enabled on your system, by reading the `/proc/config.gz` file, or listing the files in a process namespace subdirectory. For instance, listing the namespace diretory of the init process:

```
$ sudo ls /proc/1/ns
ipc  mnt  net  pid  uts  user
```

Some namespaces, such as mnt, pid, and net, require for a user to be privileged to create a child namespace.

4.3.1 mnt

The mount (MNT) namespace abstracts over the mount points of a system. This allows for processes in different namespaces to have different views of the file system. Within a container, we can unmount points that are perhaps needed by the host, but not by the container, and would perhaps make the host vulnerable, if the container had access to them.

Pivot root

One particularly useful application of mount namespaces is pivoting the file system root to some other point in the file system using [pivot_root(2)]. Pivoting the root in a container does not affect the host, or other containers. At the same time, pivoting the root moves all the dependencies on the old root, to a new root within the container.

This allows us to subsequently unmount the old root, provided that the new root does not depend on this mount point. This can be achieved by having the new root mounted as a `tmpfs`, or perhaps a read-only `squashfs`. This

hides the original root file system in a matter similar to [chroot(2)], but makes reestablishing the old root slightly more cumbersome, since the old root first has to be properly remounted first.

4.3.2 uts

The UNIX Time-sharing System (UTS) namespace abstracts over the host- and domain name of a system. This allows each container to retain a personal host- and domain name, perhaps different from the underlying host.

4.3.3 ipc

The Interprocess Communication (IPC) namespace abstracts over the IPC resources, in particular System V IPC objects and POSIX message queues. Each namespace is under the illusion that it has an isolated instance of these resources. This means that processess within one namespace cannot communicate with processes in another namespace using these primitives.

4.3.4 pid

The Process Identifier (PID) namespace abstracts over the task identifiers of a system. Tasks in different namespaces can have the same pid within their respective namespaces, but they all have distinct pid's on their hosts. Hierarchies are implemented for pid namespaces such that a host can see all the processes created within a container, while a container cannot see any of the processes on a host. Effectively, the first process in a child namespace gets pid 1, the same as an init process.

4.3.5 net

The Network (NET) namespace abstracts over the system resources associated with networking. Each network namespace has its own network devices, IP addresses, IP routing tables, port numbers, etc.

We will in general prohibit students in doing in networking. This is easy to limit with a network namespace — all networking configuration of the host is droppwed for a new child namespace.

4.3.6 user

The User (USER) namespace abstracts over the user and group ID number space. This means that the identifier of a particular user or group may differ across namespaces.

Hierarchies are implemented such that a user id in a container is mapped to a user id on the host (and likewise for groups). A container is effectively unable to list the users of a host, while a host has complete control over the capabilities of the users in the container wrt. the host system.

We can have a particular designated "container user" on the host, and map this user to UID 0 in the container. This way, even if a malicious user managed to perform a privelege escalation within the container, this would merely correspond to some unpriveleged user on the host.

This opens up containers to a wide range of capabilties, which would have otherwise required a privileged user on the host. For instance, containers can now be created using an unprivileged user in general, by creating a user namespace first.

Arguably, this leaves too much of the kernel wide open for a container, and many find that user namespaces deserver to be tested further before being enabled by default [Kerrisk (2013, Arch Linux Bug 36969, Fedora Bug 917708].

4.4 Resource Limits

The system call [getrlimit(2)], and it's sibling `setrlimit(2)` and `prlimit(2)`, can be used to manage per-user soft and hard limits on various resources. In general, when a process reaches a soft limit, it is warned, and upon reaching a hard limit it is killed, or simply prohibited in acquiring more of the resource.

An unprivileged user can freely change their soft limit to any value between 0 and the hard limit, or irreversibly lower their hard limit. A privileged user can freely change either value.

Having a designated "container" user or group, we can enforce these limits at their login using Pluggable Authentication Modules (PAM). Of particular interest, is the [pam_limits] module. We now discuss some interesting resources that we can limit, and how they can be of use.

4.4.1 nofile

The limit on the number of open files. This can be used to control how much I/O students perform and mitigate for various denial-of-service attacks.

4.4.2 nproc

The limit on the number of tasks that a user can create. This combined, with a limit on the wall-clock time can be used to mitigate for fork bombs. A fork bomb is a process that recursively creates new tasks in attempt to cause a denial-of-service.

4.5 Linux Security Modules

Linux Security Modules (LSM) is a framework that provides a mechanism for various security checks to be hooked (responded to) by new kernel extensions [Wright et al. (2002), LSM.txt]. The main use of LSM is the implementation of mandatory access control, providing a comprehensive security policy.

4.5.1 SELinux

4.5.2 AppArmor

4.5.3 Capabilities

4.6 Seccomp

Chapter 5

Assessment Pipeline

/ War is peace. Verbosity is silence. MS_VERBOSE is deprecated. */*
— DAVID HOWELLS, Linux Source Code (2012)

When a student makes a submission, it needs to be assessed. The process of assessment is specified by the teaching staff, and may consist of a series of steps — an assessment pipeline.

Each step of the pipeline is a computer program exposing a particular interface. The combination of these programs forms a pipeline, taking the student submission in at one end, and producing "student-friendly" feedback at the other.

A student submission consists of a series of files. Some of these files may be expected to specify computer programs. Let a static assessment be an assessment where we analyse the submitted files, without executing what we expect to be student programs. Let a dynamic assessment be an assessment where we attempt to execute these programs.

The primary form of assessment is static assessment. A static assessment will often facilitate a subsequent dynamic assessment. For instance, by ensuring that the student has submitted all relevant files, or by compiling student programs. Although the distinction between static and dynamic assessment is sometimes rather subtle, we provide this abstraction, and leave it to the user to judge what belongs where, according to the problem at hand.

The direct output of a static or dynamic assessment may bee to obscure or too detailed for a student to benefit from it (see also § 2.1.2/10). We introduce a feedback refinement process, intended for transforming this "raw" feedback into "student-friendly" feedback. This refinement process may further be parametrized by the current student and course progress.

S/D Asessment

Chapter 6

Infrastructure

Тяжело в учении, легко в бою; легко в учении, тяжело в бою.
(Tough in training, easy in battle; easy in training, tough in battle.)
— ALEXANDER SUVOROV, Generalissimo of the Russian Empire (1729–1800)

This chapter provides a high level overview of the chosen system architecture. We justify our choices, as well as discusses some of their benefits and downsides.

Overall, there are three types of servers involved: a key server, a Git server, and a test server. The communications between them and the students and teaching staff are roughly illustrated by Figure 6.1/37. Each type of server may in reality be a set of interconnected servers balancing loads among each other, but fulfilling together the role of a single server type.

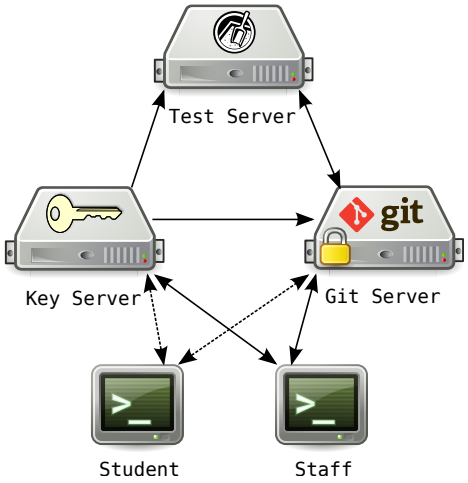

Figure 6.1: Illustration of some of the system architecture. All communication happens over secure channels. Both students and staff interact with the key server and Git server, but students have considerably less permissive rights on each of the servers

6.1 Motivation

Our current approach keeps students and staff close to the command line (as opposed to e.g. using a web-based interface). The reason is part idealogical and part practical. We believe that students become better programmers when kept close to the command line. At the same time, it is easier to implement a comprehensive information security policy in a command line bound system. A myriad of free and open source command line tools exist for assuring integral and authentic data interchange.

In particular, as an extra security precaution, we choose to encrypt student data, and have students digitally sign their data. This way, even if unauthorised users do gain access to the data, they still have to break the encryption to see the data, or fake a digital signature to tamper with it. For now, it is hard to imagine, let alone implement, such precautions in a web-based interface.

In the following subsections we briefly justify our choice of public key cryptography, and discuss various technologies that we can use to implement our cryptographic scheme.

6.1.1 Public Key Cryptography

6.1.2 OpenSSL

OpenSSL is an open source implmentation of the SSL and TLS protocols, providing at the same time a comprehensive general purpose cryptography library [openssl.org (2014)]. Despite popular belief, the `openssl(1)` command-line utility can be used to encrypt and sign regular files.

6.1.3 OpenPGP

6.2 Key Server

The purpose of the key server is to provide a centralized registry of the public keys of students and staff. The key server is not intended to be a general-purpose public key server. Instead, it's intended to serve as a certificate authority in a formal or semi-formal sense (see also § 6.3.8/45).

6.2.1 Student Key Registration

An important role of the key server is to provide the means of registering student public keys, and validating that the keys belong to the said students.

In a campus-based teaching environment, the teaching institution can serve as a validating authority. It is however, important that the registration and validation procedures do not inhibit teaching. A registration bot may be set up to aid in the matter. Let the bot be an agent with some designated private/public key pair.

A teaching institution typically designates every student with a unique identifier. In an introductory lecture, the students may be presented with a

fresh secret key, and asked to cite this secret key in an email to the bot, together with their unique identifier and public key.

The message may be additionally encrypted with bot's public key, and signed by the student's private key. This ensures that other students (and outsiders) cannot peek at a student's identity, and that a student provides a valid public key. A script can be provided to assist students in the matter.

The bot may first check the student identifier against a list of course participants, provided by the teaching staff. If authorised, the bot associates the student identifier with the email address and public key on the key server. We can now engage in secure communication with the student.

In the unlikely event of impersonification, a member of the teaching staff can go in and override this association with proper student credentials, or delete the record entirely. We trust that a campus-based teaching institution is capable of validating student identities on site. After all, if students are not technically able to participate in a course, they can be expected to eventually contact the teaching staff.

The key server can retain the students' public keys across courses, and throughout their education. It would be a great contribution to the overall web of trust on the Internet, if the bot also signed the public keys and published them to a general-purpose public key server. Having obtained appropriate permission from the students, of course.

6.2.2 Staff Key Registration

Staff key registration can be left to more manual means. It is important that staff public keys are validated in person.

To be continued..

6.2.3 Serving Public Keys

The public key registry can serve as a basis for secure communication between the students and staff in general. The public key registry should therefore be made available on the teaching institution intranet, or even to the general public.

To be continued..

6.2.4 Discussion

To be continued..

6.3 Git Server

The purpose of the Git server is to serve as a general purpose data store for both course content, assignments, and student submissions. It serves as a gateway between students and teaching staff, allowing for teaching staff to publish course content and assignments, and for students to make submissions.

In the following we refer to student and staff collectively simply as clients, connecting to our host, the Git server. In § 6.3.8/46 we will discuss why we have just one Git server.

6.3.1 Why Git?

Git is a popular [Ohloh (2014)], free, and open source distributed version control and source code management system [Git (2014)]. Although perhaps not the ideal system for all intents and purposes, it is an excellent example that has cemented itself in both the open source community, academia and industry [GitProjects (2014)].

Version control and code review are some of the Core- Tier1 and Tier2 elements in [CS Curricula 2013]. They are highly suggested topics for any undergraduate Computer Science programme.

We hypothesize that using Git for programming assignments can spur the learning of some of the workflow of modern software development. Ideally, students collaborate on assignments, while teaching staff offer code reviews, all as if it were a real software development project.

Git with authentication over SSH is an easy way to provide a scalable, online general purpose data store and gateway, having fine-grained and reliable authentication and authorisation procedures.

6.3.2 Course as a Repository

A Git server manages Git repositories. We choose to let a course be represented by a Git repository.

A Git repository has one or more branches. We choose to let one branch - the master branch - be used for the distribution of course content and assignments by teaching staff. To make submissions, students create branches in their name and push their changes to these branches onto the server.

Assessment of a student submission is provided in a special subdirectory on their private branch. All assessment is bound to particular commit by the student to a student branch.

In such an infrastructure it is important that students are not allowed to push to the master branch, or to other student branches. At the same time, teaching staff should be allowed to push to both the master (to provide content and assignments) and student branches (to provide feedback). Last but not least, we would like to let everyone see course content and assignments, but prohibit them in seeing student submissions, or pushing to any of the branches.

Such fine-grained authentication and authorisation can be achieved through OpenSSH and Git hooks.

6.3.3 OpenSSH

OpenSSH is a free (as in free speech) version of the SSH connectivity tools [openssh.com (2014)]. The tools provide for secure encrypted communication between untrusted hosts over an insecure network [ssh(1)]. They include tools for user authentication, remote command execution, file management, etc.

An OpenSSH host maintains a private/public key pair used to identify the host. Upon connection, the host offers its public key to the client, in hope that the client will accept it and (securely) proceed with authentication with the host. If authenticated, the client is mapped to a particular user on the host. After some session preparation, the client, as that user, can start a session, i.e. request a shell or the execution of a command.

One of the authentication methods supported by OpenSSH is using public key cryptography. The idea is that each client creates a private/public key pair, and informs the host of the public key over some otherwise secure channel, e.g. using a trusted keyserver.

For any user on the host, a file can be created, e.g. `~/.ssh/authorized_keys`, listing the public keys of those private/public key pairs that may be used to authenticate as that user. The format of this file [sshd(8)], allows to specify additional options for each key. The options can be used to e.g. set a session-specific environment variable, or replace the command executed once the user is authenticated. The original command is then saved as the environment variable `SSH_ORIGINAL_COMMAND`.

When using a Git server with OpenSSH, Git operations on the client, will attempt execute Git operations on the host. Per-key options can be used to make their execution dependent on the key used for authentication, e.g. performing authorisation.

6.3.4 Git Hooks

Git hooks is a Git mechanism for executing custom scripts when important events happen [git-hooks(5)]. The scripts can control in how far certain Git operations succeed. A Git hook is an adequately named executables placed in a special subdirectory in the local Git repository. Git hooks are not part of the version-controlled code base.

For instance, the `update` hook is executed whenever the client attempts to push something to a branch. The client has already been authenticated, but no changes have yet been made. The hook is passed adequate arguments to identify the branch or tag being updated and the update taking place. If this hook exits with a non-zero exit value, the update will duly fail.

6.3.5 Users

When using a Git server with OpenSSH, clients must be mapped to users on the host. There are at least two options for the mapping: each client gets their own user, or all clients map to the same user. The first option has a higher administration costs, but gives perhaps more fine grained access control.

The second option is generally more popular because of less cluttering of the UTS namespace. Additional tools, like gitolite, are instead used to provide a fine-grained access control layer. We too, have chosen this option.

6.3.6 Gitolite

Gitolite is an access control layer on top of Git [gitolite.com (2014a)]. Gitolite leverages the features of OpenSSH and Git hooks, as discussed above, to provide fine-grained authentication and authorisation [gitolite.com (2014b)].

Gitolite is used in multiple communities with high-stakes projects, such as Fedora, KDE, Gentoo, and kernel.org [gitolite.com (2014c)]. Among the reasons for choosing gitolite, kernel.org lists [kernel.org (2014)] "well maintained and supported code base", "responsive development", "broad and diverse install base", and "had undergone an external code review" [gitolite Google Group (2011)].

There are also other tools out there, such as Gerrit[1] and Stash[2]. Both of these provide a lot more than a simple access control layer.

In conclusion, we chose to use gitolite ahead of both using other tools, and implementing our own solution.

Installation

Gitolite is installed on a per-user basis. Meaning that we should create and log in as some designated Git user to set up gitolite, or change ownership accordingly after install. As an extra security assurance, the gitolite installation does not require a privileged user, so long as Git, OpenSSH, and Perl are already installed.

The code is distributed under a GNU General Public License, and is available at `git://github.com/sitaramc/gitolite`. We may wish to check out the latest tag (version), after verifying that it indeed was signed by Sitaram Chamarty (the original developer of Gitolite)[3]. To the best of our knowledge, his public GPG key is:

```
pub   4096R/088237A5 2011-10-25
      Key fingerprint =
        560A DA64 7542 816F 412E 5891 A442 9085 0882 37A5
uid      Sitaram Chamarty (work email) <sitaram@atc.tcs.com>
uid      Sitaram Chamarty <sitaramc@gmail.com>
sub   4096R/8AC76EFB 2011-10-25
```

Once cloned and compiled, gitolite setup requires the administrator's public SSH key to be provided in some accessible file:

```
$ ./gitolite setup -pk admin.pub
```

This initializes a git repository `gitolite-admin.git`, which `admin` has complete control over. This repository serves as the primary administrative interface for the gitolite access control layer.

[1] See also `https://code.google.com/p/gerrit/`.
[2] See also `https://www.atlassian.com/software/stash`.
[3] See also `http://git-scm.com/book/en/Git-Basics-Tagging`, if you are unfamiliar with Git's tagging mechanism.

Administration

Administration of the gitolite happens through a special Git repository. There are three important elements to this repository:

1. The `./keys` subdirectory which contains the public keys of all users of the system, thereby defining the users. The names of the key files designate the user names [gitolite.com (2014d)].
2. The `./conf/gitolite.conf` configuration file. This file defines the repositories, and the users' permissions wrt. those repositories.
3. The `post-update` Git hook on the server side, parsing the above keys subdirectory and config file, making adequate changes to the server repositories.

It is important that access to this repository is safely guarded as it gives complete control over the users and repositories on the Git server.

Permissions

Permissions in gitolite are granted on a per repository basis. Every time a user attempts to perform a read or write operation on a repository, the user's action is matched against a series of rules. If none of the rules match, the user operation is denied.

Permissions may be granted to the entire repository or just to a particular branch, tag, or even subfolder. Users may be granted, read, write, read-write, and even forced write permissions (more on this in the next section). There are some even more fine grained permissions [gitolite.com (2014e)], but we will not be concerned with them here.

6.3.7 Attack surface

Login shell

It is often important with Git servers to disallow clients' shell requests. This is typically achieved by setting the user's login shell to something non-permissive, e.g. a [git-shell(1)].

This set up is perhaps a bit superfluous, as gitolite disables interactive shell login via the authorized keys file. Never-the-less it is a good extra level of security, as the login shell of any user can only be modified by a privileged user, which the `git` user is not.

Session preparation dialog

When a client is authenticated with OpenSSH, but before a user session starts, the client and the host enter into a session preparation dialog.

The client can request a pseudo-tty (e.g. interactive shell), forwarding X11 connections (e.g. remote desktop), forwarding TCP connections (e.g. virtual private networking), or forwarding the authentication agent connection over

the secure channel (e.g. using the secure connection to establish other secure connections).

All these options open up the attack surface of our Git server. Fortunately, all of these session dialog options can be disabled for any key in the authorized keys file [sshd(8)]. By default, gitolite disables all of these options for all keys.

Forced push and rewriting history

Git has a, somewhat controversial [Torvalds (2007),Hamano (2009),Rego (2013)], forced push feature. This bypasses the check that the remote ref being updated should be an ancestor of the local ref used to overwrite it [git-push(1)]. Meaning that the branch being updated should be the strict base of the update.

Forced push is dangerous because it incautiously overwrites history and can thereby inhibit assessment or even modify student records.

This is mitigated for by gitolite permissions. Students are simply not allowed to perform a forced push[4]. This means that students cannot e.g. ammend to a commit that they have already pushed to the server. The students are encouraged to use [git-revert(1)] instead.

Git, OpenSSH, and Perl

Despite its popularity, relatively few vulnerabilities have ever been found outside of the Git development team [cvedetails.com (2014a)].

The security of Git (out of the box) however, depends on the sensibility of the developers involved. Impersonification and private key leaks are not always well guarded against [Gerwitzh (2013)], especially with the advent of modern Git hosting services [Homakov (2012),Huang (2013),Homakov (2014)]. It is the purpose of our Git server to serve as a guard against impersonification. Private key leaks are to be guarded against by the students themselves.

OpenSSH has also had relatively few vulnerabilities discovered outside of the OpenSSH development team [cvedetails.com (2014b)]. However, the underlying OpenSSL has been a lot less fortunate [cvedetails.com (2014c)].

Perl has only been a bit less fortunate [cvedetails.com (2014d)].

The referenced material does not cover all of the underlying libraries of the software. However, all of the above are popular pieces of software on public facing web servers. Their security therefore, is a matter of grave public concern.

6.3.8 Discussion

Pull requests

Our model of a student submission being a Git push to a student branch is not an accurate model of modern software development. In modern software development, a developer may work on their own branch (as our students would), and then make a "pull request" to merge their changes into the master branch. (Alternatively, a developer might work on in their own repository, and

[4] As an experimental bug, teaching staff are still allowed to perform a forced push.

then make a pull request for their changes to be merged into the main project repository [Bird et al. (2009].)

Such pull requests make little sense in education where all students are working on the same problem — a scenario you'd often go to great lengths to avoid in industry. Instead, students are always allowed to submit what they have to their own branch. Code reviews are then done of snapshots of the student branch, e.g. automatically every time they push, or by a human at a nominal point in time.

Alternatively, we could have chosen to have students make a pull request to a special "submission branch", with the other branch being a "draft branch". This would demand a more complicated set up of the Git server, perhaps using Gerrit or Stash, as mentioned above. The pull request could then be accepted if the code passed automatic code review.

Unfortunately, it is sometimes instructional to give credit for an attempt at solving the problem. There may even come a situation where the student has made it to submit some basic working code in the submission branch but has a more comprehensive (non-working) solution in their draft branch. It would seem that this would gravely complicate matters for the subsequent human code review. In our model, the commit and test history of a branch is sufficient to reveal when the code had last worked.

Responding to students

Responding to students via a subdirectory in their private branch means that the students have to pull from their branch before they can make a subsequent submission (the race condition aside). This is good because it encourages students to read feedback and not to push in the blind. This is bad because it might inhibit quick (re)submissions (made within minutes): as practice shows, this is frequent close to a deadline.

An alternative could be to distribute feedback in a separate private student branch, which is not writable by students. This is easy to set up in gitolite, but is more permissive of students pushing in the blind, ignoring all feedback. It also adds to the complexity of the student's view of the system: some students may fail to realise that feedback is being given at all.

As another alternative, feedback could be provided interactively, as part of a Git push operation. For instance, in a `post-receive` or a `post-update` Git hook. These hooks lets us run custom scripts after the real work of a Git push is done. The benefit is that the connection to the client is not closed until these scripts end, and standard output and error are redirected to the client [git-hooks(5)]. Although this allows to present the test results immediately, it is unclear where the test results should be persisted.

SSH Certificate Authority

Our choice of gitolite as the access control layer for our Git server, seemingly prohibits the use of an SSH certificate authority.

An SSH certificate authority is a separate server that certifies client public keys. This relinquishes a Git server of the need to keep public keys in an autho-

rized keys file, and allows to keep a centralized (hiearchical) registry of client keys. Gitolite relies on the use of an authorized keys file.

Certificate authorities however, still have to forward client certificates to the Git server. If forwarded on-demand, the certificate authority is a single point of failure. If forwarded on occasion, certificate authorities are functionally equivalent to using a Git repository over SSH, as with gitolite. We therefore do not find this to be an inconvenience.

Scalability

We dedicate a Git branch to every student. To our knowledge, there is no practical limit on the number of branches in a Git repository. If there is a limit, it has to do with underlying file system limits, as every branch requires a separate file in a particular subdirectory. This limit should be in the manner of millions, and so not applicable in a course, or department context.

Each time a client performs a Git operation, a connection to the Git server is established. The server performs one of a limited set of (presumably, finite) operations in a separate user session.

To our knowledge, there are no practical limits on the number of simultaneous connections to a Linux server. The number of user sessions however, is bound by the maximum number of processes per user. (Later, we will use this feature to guard against fork bombs.) This limit can be found using:

```
$ ulimit -u
```

Although this limit can be lifted, there is a more fundamental limit on the total number of processes for a Linux system. This limit is typically 32768. It should be increased with caution, and only if there are sufficient system resources. The limit can be found using:

```
$ cat /proc/sys/kernel/pid_max
```

A dedicated Git server should safely scale to a course, or a department, provided sufficient memory, CPU, and disk resources and speeds. On a university scale, it is advisable to use a Git server per department. OpenSSH operations are fairly CPU intensive, and many simultaneous submissions may lead us to hit the process number limits.

6.4 Encrypted and Signed Git Repositories

Although unrelated to server infrastructure, this section is included in this chapter as it is an important matter of the overall system infrastructure: this section argues for a method of securely storing, digitally signed client data.

Git has built-in a method of signing tags (versions) or commits. Both require a GPG signature.

To be continued..

6.5 Test Server

The purpose of the of the test server is to test student submissions.

To be continued..

Bibliography

[Wheeler, 2007] David A. Wheeler. *Why Open Source Software / Free Software (OSS/FS, FLOSS, or FOSS)? Look at the Numbers!*. Revised as of April 16, 2007. Retrieved from http://www.dwheeler.com/oss_fs_why.html on March 7, 2014.

Archived by WebCite® at http://www.webcitation.org/6NtSnvALX.

[CS2013] ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computer Science Curricula 2013*. December 2013. ACM Press and IEEE Computer Society Press. Retrieved from http://ai.stanford.edu/users/sahami/CS2013/final-draft/CS2013-final-report.pdf on March 7, 2014.

Archived by WebCite® at http://www.webcitation.org/6NtTXPH2s.

[EHEA (1999)] Joint declaration of the European Ministers of Education. *The Bologna Declaration*. June 19, 1999. The European Higher Education Area. Retrieved from http://www.ehea.info/Uploads/Declarations/BOLOGNA_DECLARATION1.pdf on March 9, 2014.

Archived by WebCite® at http://www.webcitation.org/6Nwr8h8l7.

[CND (2004)] Sebastian Horst and Carl Winsløw. *Undervisning i blokstruktur - potentialer og risici*. April 1, 2004. DidakTips 5. Center for Natufagenes Didaktik. University of Copenhagen. Retrieved from http://www.ind.ku.dk/publikationer/didaktips/didaktips5/5.undervisningiblokstruktur-potentialerogrisicimedomslag.pdf on March 9, 2014.

Archived by WebCite® at http://www.webcitation.org/6NwjYWlBi.

[BEK 814] Ministeriet for Forskning, Innovation og Videregående Uddannelser. *Bekendtgørelse om bachelor- og kandidatuddannelser ved universiteterne (uddannelsesbekendtgørelsen)*. Ministerial Order no. 814 of December 19, 2013. Retrieved from https://www.retsinformation.dk/Forms/R0710.aspx?id=160853 on March 9, 2014.

Archived by WebCite® at http://www.webcitation.org/6NyGHXc4N.

[BEK 1520] Ministeriet for Forskning, Innovation og Videregående Uddannelser. *Bekendtgørelse om bachelor- og kandidatuddannelser ved universiteterne (uddannelsesbekendtgørelsen)*. Ministerial Order no. 1520 of December 19,

2013. Retrieved from https://www.retsinformation.dk/Forms/R0710.aspx?id=160853 on March 9, 2014.

Archived by WebCite® at http://www.webcitation.org/6Nx4DxLGL.

[BEK 666] Ministeriet for Forskning, Innovation og Videregående Uddannelser. *Bekendtgørelse om eksamen og censur ved universitetsuddannelser (eksamensbekendtgørelsen)*. Ministerial Order no. 666 of June 24, 2012. Retrieved from https://www.retsinformation.dk/Forms/R0710.aspx?id=142560 on March 10, 2014.

Archived by WebCite® at http://www.webcitation.org/6NyI0DlqJ.

[BEK 250] Ministeriet for Forskning, Innovation og Videregående Uddannelser. *Bekendtgørelse om karakterskala og anden bedømmelse ved universitetsuddannelser (karakterbekendtgørelsen)*. Ministerial Order no. 250 of March 15, 2007. Retrieved from https://www.retsinformation.dk/Forms/R0710.aspx?id=29307 on March 11, 2014.

Archived by WebCite® at http://www.webcitation.org/6NzVOdWtv.

[Curricula (2013)] *The shared section of the BSc and MSc curricula for study programmes at the Faculty of Science, University of Copenhagen*. September 2013. Retrieved from http://www.science.ku.dk/studerende/studieordninger/faelles_sto/Faelles-del-ENG-2013-web.pdf/ on March 10, 2014.

Archived by WebCite® at http://www.webcitation.org/6NyEpPK67.

[Bradfoot & Black (2004)] Patricia Broadfoot and Paul Black. *Redefining assessment? The first ten years of assessment in education*. 2004. Assessment in Education: Principles, Policy & Practice, Vol. 11, No. 1, pp. 7–26. DOI: 10.1080/0969594042000208976. Retrieved from https://cmap.helsinki.fi/rid=1G5ND18R4-1QLJN7R-1SB on March 16, 2014.

Archived by WebCite® at http://www.webcitation.org/6O7OhF8LW.

[Pishghadam et al. (2014)] Reza Pishghadam, Bob Adamson, Shaghayegh Shayesteh Sadafian, and Flora L. F. Kan. *Conceptions of assessment and teacher burnout*. 2014. Assessment in Education: Principles, Policy & Practice, Vol. 21, No. 1, pp. 34–51. DOI: 10.1080/0969594X.2013.817382.

[Harlen & James (1997)] Wynne Harlen and Mary James. *Assessment and Learning: differences and relationships between formative and summative assessment*. 1997. Assessment in Education: Principles, Policy & Practice, Vol. 4, No. 3, pp. 365–379. DOI: 10.1080/0969594970040304.

[Butler (1988)] Ruth Butler. *Enhancing and understanding intrinsic motivation: the effects of of task-involving and ego-involving evaluation on interest and performance*. February 1988. British Journal of Educational Psychology, Vol. 58, No. 1, pp. 1–14.

[Sadler (1989)] D.Royce Sadler. *Formative assessment and the design of instructional systems*. June 1989. Instructional Science, Vol. 18, No. 2, pp. 119–144. Kluwer Academic Publishers. DOI: 10.1007/BF00117714

[Bloom et al. (1971)] Benjamin S. Bloom, J. Thomas Hastings, and George F. Madaus. *Handbook on Formative and Summative Evaluation of Student Learning*. 1971. McGraw-Hili, Inc. United States. Library of Congress Catalog Card Number 75129488. ISBN 0070061149.

[Ramaprasad (1989)] Arkalgud Rapaprasad. *On the definition of feedback*. January 1989. Behavioural Science, Vol. 28, No. 1, pp. 8–13.

[Black & William (1998)] Paul Black and Dylan William. *Assessment and Classroom Learning*. 1998. Assessment in Education: Principles, Policy & Practice, Vol. 5, No. 1, pp. 7–74. DOI: 10.1080/0969595980050102.

[Gibbs & Simpson (2004)] Graham Gibbs and Claire Simpson. *Conditions Under Which Assessment Supports Students' Learning*. May 2004. Learning and Teaching in Higher Education, Issue 1. Retrieved from http://www2.glos.ac.uk/offload/tli/lets/lathe/issue1/issue1.pdf on March 22, 2014.

Archived by WebCite® at http://www.webcitation.org/6OGkhvGyE.

[Ramsden (1992)] Paul Ramsden. *Learning to Teach in Higher Education*. 1992. Routledge. ISBN 0-415-06415-5.

[Conole & Warburton (2005)] Gráinne Conole and Bill Warburton. *A review of computer-assisted assessment*. March 2005. The Journal of the Association for Learning Technology (ALT-J), Research in Learning Technology, Vol. 14, No. 1, pp. 17–31. Retrieved from http://www.researchinlearningtechnology.net/index.php/rlt/article/download/10970/12674 on March 23, 2014.

Archived by WebCite® at http://www.webcitation.org/6OIAQzo2d.

[Valenti et al. (2003)] Salvatore Valenti, Francesca Neri, and Alessandro Cucchiarelli. *An Overview of Current Research on Automated Essay Grading*. January 2003. Journal of Information Technology Education, Vol. 2, No. 1, pp. 319–330.

[Ala-Mutka (2005)] Kirsti M. Ala-Mutka. *A Survey of Automated Assessment Approaches for Programming Assignments*. June 2005. Computer Science Education, Vol. 15, No. 2, pp. 83–102.

[Bull & McKenna (2004)] Joanna Bull and Colleen McKenna. *Blueprint for Computer-Assisted Assessment*. 2004. Taylor & Francis e-Library. Master e-book ISBN: 0-203-46468-0.

[Topping (1998)] Keith Topping. *Peer Assessment between Students in Colleges and Universities*. Autumn 1998. Review of Educational Research, Vol. 68, No. 3, pp. 249–276.

[Carter et al. (2003)] Janet Carter, John English, Kirsti Ala-Mutka, Martin Dick, William Fone, Ursula Fuller, Judy Sheard. *How Shall We Assess This?* December 2003. Working group reports from ITiCSE on Innovation and technology in computer science education (ITiCSE-WGR '03), David Finkel (Ed.), pp. 107–123. DOI: 10.1145/960492.960539.

[CS Curricula 2013] The Joint Task Force on Computing Curricula. Association for Computing Machinery (ACM). IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. December 20, 2013. Retrieved from http://www.acm.org/education/CS2013-final-report.pdf on March 29, 2014.

Archived by WebCite® at http://www.webcitation.org/6ORGeUYy5.

[Skinner (1965)] Burrhus Frederic Skinner. Harvard University. *Reflections on a Decade of Teaching Machines*. 1965. In Teaching Machines and Programmed Learning, Vol. 2, Robert Glaser (Ed.), pp. 5–20. National Education Association of the United States. LCCN: 60-15721.

[Kember (1997)] David Kember. Hong Kong Polytechnic University. *A Reconceptualisation of the Research into University Academics' Conceptions of Teaching*. 1997. Learning and Instruction, Vol. 7, No. 3, pp. 225–275.

[Sclater & Howie (2003)] Niall Sclater and Karen Howie. Center for Educational Systems, University of Strathclyde, Scotland, UK. *User requirements of the "ultimate" online assessment engine*. April 2003. Computers & Education, Vol. 40, No. 3, pp. 285–306. DOI: 10.1016/S0360-1315(02)00132-X.

[Ohloh (2014)] Ohloh. Black Duck Software, Inc. Tools. Compare Repositories. Retreived from http://www.ohloh.net/repositories/compare on April 12, 2014.

[Git (2014)] Git. *Git and Software Freedom Conservancy*. Retrieved from http://git-scm.com/sfc on April 12. 2014.

Archived by WebCite® at http://www.webcitation.org/6OmayfnSi.

[GitProjects (2014)] GitProjects. *Projects that use Git for their source code management*. Git Wiki. Last updated on April 5, 2014. Retrieved from https://git.wiki.kernel.org/index.php/Main_Page on April 12, 2014.

Archived by WebCite® at http://www.webcitation.org/6OmbdAIQF.

[gitolite.com (2014a)] *Hosting git repositories*. Retrieved from http://gitolite.com/gitolite/index.html on Arpil 13, 2014.

Archived by WebCite® at http://www.webcitation.org/6OnsR9f7R.

[gitolite.com (2014b)] *authentication and authorization in gitolite*. Retrieved from http://gitolite.com/gitolite/how.html on Arpil 13, 2014.

Archived by WebCite® at http://www.webcitation.org/6OoAvdLeK.

[gitolite.com (2014c)] *who uses gitolite*. Retrieved from http://gitolite.com/gitolite/who.html on Arpil 13, 2014.

Archived by WebCite® at http://www.webcitation.org/6OoB1QVpA.

[gitolite.com (2014d)] *adding and removing users*. Retrieved from http://gitolite.com/gitolite/users.html on Arpil 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvHM2Z2h.

[gitolite.com (2014e)] *different types of write operations*. Retrieved from http://gitolite.com/gitolite/write-types.html on Arpil 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvIKAXXm.

[kernel.org (2014)] *How does kernel.org provide its users access to the git trees?* Frequently asked questions. The Linux Kernel Archives. Retrieved from https://www.kernel.org/faq.html#whygitolite on April 13, 2014.

Archived by WebCite® at http://www.webcitation.org/6OoBgEPwG.

[gitolite Google Group (2011)] Dan Carpenter and Sitaram Chamarty. *security audit of Gitolite*. Discussion on the gitolite Google Group. First post made on September 30, 2011. Last post made on October 2, 2011. Retrieved from https://groups.google.com/d/topic/gitolite/jcUkIFKxbQ8/discussion on April 13, 2014.

Archived by WebCite® at http://www.webcitation.org/6OoCHMGop.

[openssh.com (2014)] OpenBSD. OpenSSH. Retrieved from http://www.openssh.com/ on April 16, 2014.

Archived by WebCite® at http://www.webcitation.org/6OsQXs51M.

[openssl.org (2014)] OpenSSL. Cryptography and SSL/TLS Toolkit. Retrieved from https://www.openssl.org/ on April 20, 2014.

Archived by WebCite® at http://www.webcitation.org/6OyQjYQnz.

[ssh(1)] BSD. *SSH(1)*. ssh — OpenSSH SSH client (remote login program). BSD General Commands Manual. Published April 6, 2014. Retrieved from http://man7.org/linux/man-pages/man1/ssh.1.html on April 16, 2014.

Archived by WebCite® at http://www.webcitation.org/6OsTPwEgi.

[sshd(8)] BSD. *SSHD(8)*. sshd — OpenSSH SSH daemon. BSD System Manager's Manual. Published on April 6, 2014. Retrieved from http://man7.org/linux/man-pages/man8/sshd.8.html on April 16, 2014.

Archived by WebCite® at http://www.webcitation.org/6OsTLo4Ce.

[git-shell(1)] Git 1.9.rc1. *GIT-SHELL(1)*. git-shell - Restricted login shell for Git-only SSH access. Git Manual. Published January 30, 2014. Retrieved from http://man7.org/linux/man-pages/man1/git-shell.1.html on Arpil 16, 2014.

Archived by WebCite® at http://www.webcitation.org/6OsZP6SgC.

[git-push(1)] Git 1.9.rc1. *GIT-PUSH(1)*. git-push - Update remote refs along with associated objects. Git Manual. Published January 30, 2014. Retrieved from http://man7.org/linux/man-pages/man1/git-push.1.html on Arpil 16, 2014.

Archived by WebCite® at http://www.webcitation.org/6Ot2V2qp1.

[git-revert(1)] Git 1.9.rc1. *GIT-REVERT(1). git-revert - Revert some existing commits.* Git Manual. Published January 30, 2014. Retrieved from http://man7.org/linux/man-pages/man1/git-revert.1.html on Arpil 17, 2014.

Archived by WebCite® at http://www.webcitation.org/6Ot6LflKS.

[git-hooks(5)] Git 1.9.rc1. *GITHOOKS(15). githooks - Hooks used by Git.* Git Manual. Published January 30, 2014. Retrieved from http://man7.org/linux/man-pages/man5/githooks.5.html on Arpil 17, 2014.

Archived by WebCite® at http://www.webcitation.org/6OtvjunoB.

[clone(2)] Linux. *CLONE(2). `clone`, `__clone2` - create a child process*. Linux Programmer's Manual. Published February 27, 2014. Retrieved from http://man7.org/linux/man-pages/man2/clone.2.html on April 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6P04xS6aU.

[unshare(2)] Linux. *UNSHARE(2). `unshare` - disassociate parts of the process execution context*. Linux Programmer's Manual. Published April 17, 2013. Retrieved from http://man7.org/linux/man-pages/man2/unshare.2.html on May 8, 2014.

Archived by WebCite® at http://www.webcitation.org/6PPQVEMzp.

[setns(2)] Linux. *SETNS(2). `setns` - reassociate thread with a namespace*. Linux Programmer's Manual. Published January 1, 2013. Retrieved from http://man7.org/linux/man-pages/man2/setns.2.html on May 8, 2014.

Archived by WebCite® at http://www.webcitation.org/6PPQyqMeY.

[fork(2)] Linux. *FORK(2). `fork` - create a child process*. Linux Programmer's Manual. Published March 12, 2014. Retrieved from http://man7.org/linux/man-pages/man2/fork.2.html on April 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6P065nmSn.

[proc(5)] Linux. *PROC(2). `proc` - process information pseudo-filesystem*. Linux Programmer's Manual. Published April 12, 2014. Retrieved from http://man7.org/linux/man-pages/man5/proc.5.html on April 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6P0AnUQsS.

[pivot_root(2)] Linux. *PIVOT_ROOT(2). `pivot_root` - change the root filesystem*. Linux Programmer's Manual. Published June 13, 2013. Retrieved from http://man7.org/linux/man-pages/man5/pivot_root.2.html on May 12, 2014.

Archived by WebCite® at http://www.webcitation.org/6PVmriIoe.

[chroot(2)] Linux. *CHROOT(2). `chroot` - change root directory*. Linux Programmer's Manual. Published September 20, 2010. Retrieved from http://man7.org/linux/man-pages/man5/chroot.2.html on May 12, 2014.

Archived by WebCite® at http://www.webcitation.org/6PVn2lIwz.

[getrlimit(2)] Linux. *GETRLIMIT(2). `getrlimit`, `setrlimit`, `prlimit` - get/set resource limits*. Linux Programmer's Manual. Published January 22, 2014. Retrieved from http://man7.org/linux/man-pages/man5/getrlimit.2.html on May 13, 2014.

Archived by WebCite® at http://www.webcitation.org/6PXC3h35y.

[pam_limits] Cristian Gafton. *6.15. `pam_limits` - limit resources*. The Linux-PAM System Administrators' Guide. Version 1.1.2, 31. August 2010. Retrieved from http://www.linux-pam.org/Linux-PAM-html/sag-pam_limits.html on May 13, 2014.

Archived by WebCite® at http://www.webcitation.org/6PXDQwwd6.

[Bird et al. (2009] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. *The promises and perils of mining git*. Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories, pp. 1–10. IEEE Computer Society Washington, DC, USA. ISBN 978-1-4244-3493-0.

[Torvalds (2007)] Linus Torvalds. *Re: Modify/edit old commit messages*. Retrieved from http://www.gelato.unsw.edu.au/archives/git/0702/38650.html on April 17, 2014.

Archived by WebCite® at http://www.webcitation.org/6Ot2yqQYR.

[Hamano (2009)] Junio C Hamano. In response to "*How do I push amended commit to the remote git repo?*" StackOverflow. Posted on January 11, 2009, under the username "gitster". Edited by Gottlieb Notschnabel on September 3, 2013. Retrieved from http://stackoverflow.com/a/432518/108100 on April 17, 2014.

Archived by WebCite® at http://www.webcitation.org/6Ot4dU85N.

Revision history retrieved from http://stackoverflow.com/posts/432518/revisions on April 17, 2014.

Archived by WebCite® at http://www.webcitation.org/6Ot4hGKpY.

[Rego (2013)] Cauê C. M. Rego. In response to "*How to properly force a Git Push?*" StackOverflow. Posted on May 22, 2013, under the username "Cawas". Retrieved from http://stackoverflow.com/a/16702355/108100 on April 17, 2014.

Archived by WebCite® at http://www.webcitation.org/6Ot4qY1Y2.

[cvedetails.com (2014a)] CVE Details. The ultimate security vulnerability datasource. *GIT: Security Vulnerabilities*. Retrieved from http://www.cvedetails.com/vulnerability-list/vendor_id-4008/GIT.html on April 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvKd3HTW.

[cvedetails.com (2014b)] CVE Details. The ultimate security vulnerability datasource. *Openssh: Security Vulnerabilities*. Retrieved from http://www.

cvedetails.com/vulnerability-list/vendor_id-7161/Openssh.html, on April 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvM6eL7m.

[cvedetails.com (2014c)] CVE Details. The ultimate security vulnerability datasource. *Openssl: Security Vulnerabilities*. Retrieved from http://www.cvedetails.com/vulnerability-list/vendor_id-217/Openssl.html, on April 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvMlTlsh.

[cvedetails.com (2014d)] CVE Details. The ultimate security vulnerability datasource. *Perl: Security Vulnerabilities*. Retrieved from http://www.cvedetails.com/vulnerability-list/vendor_id-1885/Perl.html, on April 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvMGTWjE.

[Huang (2013)] Jay Huang. *Pushing code to GitHub as Linus Torvalds*. Posted on December 16, 2013. Retrieved from http://www.jayhuang.org/blog/pushing-code-to-github-as-linus-torvalds/ on April 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvK7dQBu.

[Gerwitzh (2013)] Mike Gerwitz. *A Git Horror Story: Repository Integrity With Signed Commits*. Last edited on November 10, 2013. Retrieved from http://mikegerwitz.com/papers/git-horror-story on April 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvKR8Hzi.

[Homakov (2012)] Egor Homakov. *Hacking rails/rails repo*. Published on March 4, 2012. Retrieved from http://homakov.blogspot.dk/2012/03/how-to.html on April 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvLP3zYf.

[Homakov (2014)] Egor Homakov. *How I hacked GitHub again*. Published on February 7, 2014. Retrieved from http://homakov.blogspot.dk/2014/02/how-i-hacked-github-again.html on April 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6OvKurm25.

[Hutchings (2011)] Ben Hutchings. *Accepted linux-2.6 3.0.0~rc1-1~experimental.1 (source all amd64)*. Email sent to the Debian Mailing Lists on June 1, 2011. Retrieved from http://packages.qa.debian.org/l/linux-2.6/news/20110601T223515Z.html on April 25, 2014.

Archived by WebCite® at http://www.webcitation.org/6P7kSQe9C.

[Wright et al. (2002)] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. *Linux Security Modules: General Security Support for the Linux Kernel*. August 2002. In Proceedings of the 11th USENIX Security Symposium. USENIX Association. San Francisco, California, United States. Retrieved from https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright.pdf on April 19, 2014.

Archived by WebCite® at http://www.webcitation.org/6OxFtc1KV.

[Turner et al. (2010)] Paul Turner, Bharata B Rao, and Nikhil Rao. *CPU bandwidth control for CFS*. July 2010. In Linux Symposium, Vol 10, pp. 245–254. Ottawa, Ontoria, Canada. Retrieved from https://www.kernel.org/doc/ols/2010/ols2010-pages-245-254.pdf on April 28, 2014.

Archived by WebCite® at http://www.webcitation.org/6P9ouxLJJ.

[Kerrisk (2013] Michael Kerrisk. *Namespaces in operation, part 1: namespaces overview*. January 4, 2013. LWN.net. Retrieved from http://lwn.net/Articles/531114/ on May 12, 2014.

Archived by WebCite® at http://www.webcitation.org/6PWKMBrvj.

[Arch Linux Bug 36969] Arch Linux. *FS#36969 - [linux] 3.13 add CONFIG_USER_NS*. Arch Linux Bugtracker. Bug filed on September 17, 2013. Retrieved from https://bugs.archlinux.org/task/36969 on May 12, 2014.

Archived by WebCite® at http://www.webcitation.org/6PWL6yV41.

[Fedora Bug 917708] Fedora. *Bug 917708 - Re-enable CONFIG_USER_NS*. Red Hat Bugzilla. Bug filed on March 4, 2013. Retrieved from https://bugzilla.redhat.com/show_bug.cgi?id=917708 on May 12, 2014.

Archived by WebCite® at http://www.webcitation.org/6PWMQAOoI.

[Veldhuizen (2003)] Todd L. Veldhuizen. *C++ Templates are Turing Complete*. 2003. Indiana University of Computer Science. Retrieved from http://port70.net/~nsz/c/c++/turing.pdf on May 18, 2014.

Archived by WebCite® at http://www.webcitation.org/6Pf92rHhX.

[McCauley et al. (2008)] Renée McCauley, Sue Fitzgerald, Gary Lewandowski, et al. *Debugging: A Review of the Literature from an Educational Perspective*. June 2008. Computer Science Education Vol. 18, No. 2, pp.67–92.

[Lerner et al. (2007)] Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. *Searching for Type-Error Messages*. 2007. In Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI '07). ACM, New York, NY, USA, 425–434.

[Linux kernel (v3.14.2)] The Linux kernel v3.14.2.

Retrieved from https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.14.2.tar.xz on April 28, 2014.

Signed by Greg Kroah-Hartman, using the GPG key:

```
pub   4096R/6092693E 2011-09-23
uid     Greg Kroah-Hartman
          (Linux kernel stable release signing key) <greg@kroah.com>
sub   4096R/76D54749 2011-09-23
```

Yielding the signature:

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v2.0.22 (GNU/Linux)

iQIcBAABAgAGBQJTXE0zAAoJEDjbvchgkmk+0AUQAKrfkqeRXpwePAEHFCBqTqvN
R2fZa6tAY1w5psN8grWh2seu2W1KAtEk53oht/6uZITqs3i2pYmRAJyEzVTBggs9
4BI0rC1qebeiO3wkD1biRAIMPQWt6UAB/pvjBeMmMiw4G7FFZHSvBSct91Dsnb87
4A7083ZT/A421C2OtH3vROehyQDyfHp+oL22SKMCoXKCCMCDZp5K07AMVggrzDoZ
KGDEeBpowCSCtoUEEBlrVGz/syyaWZzzcMy+UYeZ12JxpfgnX5oq14w1HIPfAhJn
/P6x7OvmN75oIrxrt4rRs+aUY97iuiEzPpn9F2K4rNruTZUXN7906h/WWCJ/K/b0
D80wC1msaJqMYIEhQICu5kwezVswKVHz3QM9BO1ak3RgObw3j70KKVxJQ95I6jYn
I3uz8RDGXWvp+6aso8vl/HWbQ6dCCA/9plYALJZmRcy2Yg0A0nH3w6+ckC1x/r4l
ZyR6NEcVYg27HQswjmWxbqUhapFMLQGj5oGZ9svbsdwet3ckQTcqAtS5N/YHZZaQ
SnzvY4dZ/MoRwdCGz0hC99RofIgMPgY8ypkc2GGvyGv9uDLsK4koB65ZX1zW/oRw
43eatEoY/Q1QyGWrbwqEWFY91XbZne1KJNwdXYkmTDawMI2F2zApIjsAHpMseJiN
XZPAJqjjAF6nhxRzsrI8
=HZrL
-----END PGP SIGNATURE-----
```

[resource_counter.txt] Li Zefan, et al. *The Resource Counter*. Found in [Linux kernel (v3.14.2)], under `./Documentation/cgroups/`.

[cpuacct.txt] Bharata B Rao, et al. *CPU Accounting Controller*. Found in [Linux kernel (v3.14.2)], under `./Documentation/cgroups/`.

[memory.txt] Li Zefan, et al. *Memory Resource Controller*. Found in [Linux kernel (v3.14.2)], under `./Documentation/cgroups/`.

[cgroups.txt] Paul Menage, Paul Jackson, Chirstoph Lameter, et al. *CGROUPS*. Found in [Linux kernel (v3.14.2)], under `./Documentation/cgroups/`.

[devices.txt] Li Zefan, Aristeu Rozanski, et al. *Device Whitelist Controller*. Found in [Linux kernel (v3.14.2)], under `./Documentation/cgroups/`.

[blkio-controller.txt] Li Zefan, Aristeu Rozanski, et al. *Block IO Controller*. Found in [Linux kernel (v3.14.2)], under `./Documentation/cgroups/`.

[sched-design-CFS.txt] J. Bruce Fields, et al. *CFS Scheduler*. Found in [Linux kernel (v3.14.2)], under `./Documentation/scheduler/`.

[sched-bwc.txt] Bharata B Rao. *CFS Bandwidth Control*. Found in [Linux kernel (v3.14.2)], under `./Documentation/scheduler/`.

[sched-rt-group.txt] J. Bruce Fields. *Real-Time group scheduling*. Found in [Linux kernel (v3.14.2)], under `./Documentation/scheduler/`.

[kernel-parameters.txt] Linus Torvalds, et al. *Kernel Parameters*. Found in [Linux kernel (v3.14.2)], under `./Documentation/`.

[LSM.txt] Kees Cook. *Linux Security Module Framework*. Found in [Linux kernel (v3.14.2)], under `./Documentation/security/`.

Appendix A

General Linux Concepts

This chapter covers some general Linux concepts which are not necessarily to known to the casual Linux user.

A.1 Tasks

The distinction between a thread and a process in the Linux kernel is somewhat more subtle than in operating systems textbooks.

In the Linux kernel, a thread of execution, also called a task, has a thread ID (also called a PID inside the kernel), a thread group ID, and a parent thread group ID. New tasks can be created using the [clone(2)] system call[1]. Depending on the parameters passed, the child task can share various parts of its execution context with its parent task. For instance, we may choose to stay under the same thread group ID, or parent thread group ID, share open files, memory, etc.

A process is a nonempty set of tasks that share the same thread group ID. A process is identified by its thread group ID. This is what is commonly referred to as the PID in user space, whereas we refer to a thread ID as TID (unlike PID inside the kernel). The system calls `gettid(2)`, `getpid(2)` and `getppid(2)` return the thread ID, thread group ID, and parent thread group ID of the running task, respectively.

[1] The more canonical [fork(2)] system call is seldom used. Its behaviour can be mimicked by [clone(2)]. This is indeed what the standard glibc `fork()` does.