

Online TA

Master Project

Datalogisk institut, Copenhagen University (DIKU)

Oleksandr Shturmov

`oleks@oleks.info`

May 21, 2014.

Thanks to David Plassmann, for many interesting discussions on the Linux kernel internals and user space tools.

Contents

0	Preface	6
0.1	Audience	6
0.2	Dictionary and Grammar	6
0.3	Legal Disclaimer	6
0.4	References	7
0.5	About the Author	7
0.6	Remarks	7
1	Introduction	8
1.1	Assessment in Education	8
1.1.1	Categorising Assessment	8
1.1.2	Feedback	9
1.1.3	Computer-Assisted Assessment	9
1.1.4	Assignment	10
1.1.5	Submission	10
1.1.6	Rigor	10
1.2	Courses	11
1.3	Assessment in Programming Education	11
1.3.1	Programming Languages	11
1.3.2	Environments	12
1.3.3	Courses	12
1.4	Roles in Educational Assessment	13
1.4.1	Teachers	13
1.4.2	Students	14
1.4.3	General Public	14
1.5	Related Work	15
2	Analysis	16
2.1	Making Submissions	16
2.2	Static Analyses	16
2.3	Dynamic Analyses	17
2.4	Feedback	18
2.5	Course Content	19
2.6	Other Notions	19
2.6.1	Safety	19
2.6.2	Fairness	19

2.6.3	Security	20
3	Sandboxing untrusted code	21
3.1	Technology Overview	21
3.1.1	Operating system-level virtualization	22
3.2	Control Groups	23
3.2.1	Managing cgroups	24
3.2.2	The Resource Counter	25
3.2.3	memory	25
3.2.4	cpuacct	26
3.2.5	cpu	26
3.2.6	devices	27
3.3	Namespaces	28
3.3.1	mnt	28
3.3.2	uts	29
3.3.3	ipc	29
3.3.4	pid	29
3.3.5	net	29
3.3.6	user	30
3.4	Resource Limits	30
3.4.1	nofile	30
3.4.2	nproc	30
3.5	Linux Security Modules	31
3.5.1	SELinux	31
3.5.2	AppArmor	31
3.5.3	Capabilities	31
3.6	Seccomp	31
4	Assessment Pipeline	32
5	Infrastructure	33
5.1	Motivation	34
5.1.1	Public Key Cryptography	34
5.1.2	OpenSSL	34
5.1.3	OpenPGP	34
5.2	Key Server	34
5.2.1	Student Key Registration	34
5.2.2	Staff Key Registration	35
5.2.3	Serving Public Keys	35
5.2.4	Discussion	35
5.3	Git Server	35
5.3.1	Why Git?	36
5.3.2	Course as a Repository	36
5.3.3	OpenSSH	36
5.3.4	Git Hooks	37
5.3.5	Users	37
5.3.6	Gitolite	38
5.3.7	Attack surface	39

5.3.8	Discussion	40
5.4	Encrypted and Signed Git Repositories	42
5.5	Test Server	43
A	General Linux Concepts	55
A.1	Tasks	55
B	Technical Details	56
B.1	Control Groups	56
B.1.1	Managing cgroups	56
B.1.2	memory	56
B.1.3	cpu	57
B.1.4	cpuset	57
B.1.5	devices	57
B.1.6	blkio	57

Chapter 0

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.

0.1 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 computers. Preferably, the reader should hold a Computer Science degree, be, or have been enrolled in a Computer Science university programme.

0.2 Dictionary and Grammar

Unless otherwise stated, the reader

0.3 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.

0.4 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[®].

0.5 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.

0.6 Remarks

One major deficiency of the current version of this report is that it presents no concrete sample uses of the system designed. This is primarily due to time constraints. Concrete sample uses will be prepared for the defence.

Chapter 1

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. 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, discuss why a solution remains elusive, and discuss some related work. 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.

1.1 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 learning¹ [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)].

1.1.1 Categorising Assessment

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

¹Sometimes, 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.

1.1.2 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.

1.1.3 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.

1.1.4 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.

1.1.5 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.

1.1.6 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 computer programming, high rigor can often be expected in submissions. We explore this notion in the following sections.

1.2 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 encourage 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.

1.3 Assessment in Programming Education

Computer programming is concerned with the construction of programs executable 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.

More often than not, computer programs are written for others than the original authors to use, test, and maintain. Computer programming is therefore also often concerned with the construction of usable, testable, readable, and maintainable programs. To this end, for particular assignments, we may wish for students to practice following a particular style guide, or practice the use of a particular programming abstraction.

1.3.1 Programming Languages

Assignments in computer programming 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 pre-

dominant one for teaching programming. It is most useful, therefore, to facilitate assessment of programs written in any conceivable programming language.

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.

1.3.2 Environments

Assignments in computer programming will often ask students to write programs that operate within environments with particular permissions and restrictions. We wish to facilitate such permissions and ensure that the restrictions are adhered to. 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.

1.3.3 Courses

Confined to programming education, we may still hold a wide range of different courses, concerned with a particular programming language, particular programming techniques, or managing system resources. Considering which of these courses pose the biggest challenges to computer-assisted assessment, systems programming in C comes to mind.

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.

Systems programming typically also involves 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 use in the context of a real system.

Systems 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 systems programming in C, we can facilitate safe and fair environments for most courses in computer programming in general.

We will however, make a few simplifying assumptions as we seek to meet certain safety, fairness, and security requirements. For instance, we will assume that students do not need access to networking facilities, or access to more than a handful of character devices.

1.4 Roles in Educational Assessment

In this section, we expand on the roles of the teaching staff, students, and the general public.

1.4.1 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, or performing the role of external examiners.

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.

Since teachers facilitate student learning, as well as the students' final assessment, teachers exert great authority over students. Teachers decide how tough the course and its assessments are.

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 a handful of teachers.

Instead of hiring more teachers, the strategy is often to rely on some methods 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, teaching assistants, and external examiners, collectively as teaching staff, or simply, staff.

Trust

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

In various disciplines, especially in computer programming, assessment involves some fairly mechanical processes. The teaching staff 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.

They 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 teaching staff. It is important that their programming mistakes don't get in the way of student learning.

1.4.2 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 a perspective employer, acquaintance, 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 unjustly that they have acquired various knowledge or skills. 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 computer-assist assessment, can be that students can fail fast and fail often. This may facilitate learning.

1.4.3 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 to course content 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 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.

1.5 Related Work

[[Sclater & Howie \(2003\)](#)] arrive at a set of “ultimate” requirements for an online assessment engine. Their requirements represent the wishes of a focus group at a particular institution, who have worked with similar engines before. It is worth noting, that the focus group does not include students — an important user group.

Harvard has recently done a major redesign of their introductory Computer Science course, CS50 [[Malan \(2010a\)](#)]. They’ve moved to securely executing source code coming from students, and even arbitrary users on the Internet [[Malan \(2010b\)](#), [Malam \(2013\)](#)]. Many of the techniques they use, are (incidentally) also employed here. We see similar work going on at the Stanford Computer Security Lab [[Stefan \(2013\)](#)].

A related area of interest are so-called “online judges” for programming competitions. The stark difference stands in that an online judge seeks to find the *first* contestants pass an assignment, whereas an online TA aids in getting *all* students to eventually pass an assignment. The parallels can be drawn in that the incentives to use (and abuse) an “online” judge, are often similar to the incentives to use (and abuse) an “online” TA. A lot of work has been done in this area to provide for lightweight, safe, secure, and fair sandboxes for running arbitrary user code. See e.g. [[DOMJudge \(2014\)](#)].

Chapter 2

Analysis

Students make submissions in response to assignments. We would like to assess their work with the assistance of computers. We can do this by offering students a hosted service, whereby students receive course content, assignments, make submissions and receive feedback about their work.

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

2.1 Making Submissions

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. We can mitigate for this using symmetric cryptography and cryptographic signatures. This is further discussed in § 5/33.

2.2 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.

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 language¹. Furthermore, we may wish for the program to compile, assemble, etc. with particular flags.

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.

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 adheres to comprehensive, perhaps assignment-specific, style guides.

Checking that a submission meets all such requirements is usually a matter of static analysis — checking that a submission meets programming language requirements should not result in the execution of programs written by students in a Turing-complete language in general².

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.

This amounts to a range of static analyses that we may want to perform, some of them in lockstep with dynamic analyses. In case of failure of any of the analyses, feedback is generated. We intend to process this feedback before handing it back to the students for pedagogical reasons.

2.3 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.

Student programs often require some input data, and produce some output data — they may even be interactive³. For a particular assignment, we make

¹The 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.

²One exception to the rule is C++ Templates [Veldhuizen (2003)].

³For simplicity, we currently ignore assignments that require building a graphical user interface. This still leaves rich interactiveness capabilities.

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.
- (c) a subsequent input may depend on a previous output (interactiveness).

This requires for an input generator and an output validator to run in lock-step with the student program, perhaps as a single process. We'll regard them as simple interactive (stdin/stdout) processes, and leave their design to teaching staff in general.

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, 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 analyses of submissions may interfere with other analyses, causing faulty assessments. We seek to 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. Furthermore, 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.

In general, dynamic analyses generate feedback wrt. whether the student program produces the expected results, and its resource use (or abuse). A misbehaving student program may be halted prematurely. We intend to process this feedback before handing it back to the students.

2.4 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 notice. We do not consider feedback processing beyond the notion that it is an interactive (stdin/stdout) process, defined per-assignment by the teaching staff. 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, to add to it. We would like to give the teaching staff the ability to elaborate on, or discard feedback after it has already been given to the student. That is, automatic feedback is given immediately, but can be overridden by the teaching staff, yielding new feedback for the student. This way, a student can get immediate (automatic) feedback on their program, and later, (manual) feedback on their report.

This is for both a particular submission, or all submissions for an assignment at once. The latter is especially useful when an error is discovered in either the input generator, the output validator, or the feedback processor — all processes designed by the teaching staff.

2.5 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).

2.6 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 no data need ever be deleted, until an entire course is securely wiped by a system administrator. It follows that some of the data may lose integrity or become vulnerable over time.

2.6.1 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.

2.6.2 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 could be queued on a first-come-first-serve basis, provided that a student can have at most one submission assessed at a time.

2.6.3 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 § 1.4/13, 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 3

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. Computer-assisted assessment constitutes a range of static and dynamic analyses of a submission. Static analyses constitute executing computer programs specified by the teaching staff, which read and analyze student files. Dynamic analyses constitute executing, and monitoring the runtime behaviour of student programs.

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 for such “sandboxing”. We quickly 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 system resources within [[Linux kernel \(v3.14.2\)](#)], henceforth the Linux kernel.

3.1 Technology Overview

A program is executed within a program execution environment. A sandboxed execution environment ensures the non-interference of the sandboxed program with other programs on the system. There are two general approaches to sandboxing: sandboxing the operating system, or sandboxing programs within the operating system.

In § 2/16, we discussed that for each submission we would like to run a range of static and dynamic analyses, and run their feedback through a feedback processor. Furthermore, a dynamic analysis constitutes executing the student program, an input generator, and an output validator, all in lockstep with

one another. This constitutes a range of simultaneous and interactive processes involved in an assessment.

If the assessment happens all in a single sandboxed operating system, it seems infeasible to monitor, or impose limits on, e.g. the student program, independent of the rest. Alternatively, we could host each separate process in its own, sandboxed operating system. Although seemingly more useful, it adds to the complexity of communication between the processes of an assessment.

We choose to discard sandboxing the operating system, and consider sandboxing within the operating system. As we shall see, this allows for fine-grained resource monitoring and limits, while retaining simple means of interprocess communication.

3.1.1 Operating system-level virtualization

With operating system-level virtualization (OSLV), instead of running an operating system within a virtual environment, we create virtual environments within the operating system — all sandboxes run within the same kernel.

Time-sharing systems have for a long time provided for multiple simultaneous user space instances on within 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”. Similarly, user programs are “jailed”, or “contained”.

The pitfall of OSLV in general, is that we become more vulnerable to kernel vulnerabilities. If a sandboxed program can utilize a kernel vulnerability, the whole system is under threat.

FreeBSD Jails

FreeBSD’s [\[jail\(8\)\]](#) is perhaps the first and most notorious (due to naming) implementation of OSLV. The problem is that it only runs on strictly FreeBSD-based operating systems, e.g. not Mac OS X. This is impractical because FreeBSD is less popular than Linux in general, and so a lot of software, perhaps necessary for assessment, may be a hurdle to maintain.

Linux Kernel Containment

The Linux kernel, has in the recent decade acquired many of the features of FreeBSD jails, and more. Containerisation in the Linux kernel is also done in a fairly extensible fashion, providing for new extensions, abstracting over new system resources in the future.

A lot of this work is inspired by OpenVZ, an OSLV technology, based on a modified Linux kernel. Many of their changes to the kernel have been making their way into the mainstream kernel [[Kerrisk \(2012\)](#)]. Due to this gradual merge, and having more confidence in the mainstream Linux kernel, we consider only the containerisation options already available there.

LXC et al.

There exist a number of userspace interfaces to the Linux kernel containment features. We have found all these tools lacking in some respect, and so have gone with setting up our own lightweight interface¹.

Linux Containers (LXC) were originally developed by Daniel Lezcano (IBM) to ease life for kernel developers working on the kernel containerisation features [LXC (v1.0.3)]. To our knowledge, LXC is the most feature-rich, and stable userspace interface to the Linux kernel containment features [Graber (2014)].

The general approach of LXC is to let you set up containers managed by an LXC daemon. It is fairly straight forward to set up independent containers for dedicated users, or shared read-only containers.

We would like to set up containers that share many resources (e.g. a read-only root file system), but differ in a few aspects (e.g. running an individual student program), and provide for a safe file system scratchspace for students. Also, we do not intend to retain our containers (only data) after the process in question has finished. In our experience, this is fairly laborious with LXC.

There is also a range of high-level frameworks for managing Linux containers, using LXC, and other drivers. For instance, libvirt-lxc² and Docker³. These tools seem to focus on providing a sandboxed userland rather than a sandboxed execution of a program. In a way, this stems from the general LXC approach to containers, providing for managed containers, rather than managed classes of containers.

These frameworks, together with LXC, can prove useful when designing standard templates for courses. Having e.g. appropriate software already installed. We will not be concerned with this here.

3.2 Control Groups

Control groups (cgroups) provide a mechanism of hierarchically grouping/-partitioning tasks (see also Appendix A.1/55), and their children [cgroups.txt]. Some more technical details, as well as less related aspects of cgroups are discussed in Appendix B.1/56.

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”.

3.2.1 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

¹Ideally, we’d probably contribute adequate changes to e.g. the LXC project. The process has however, provided for a better understanding of this part of the Linux kernel.

²See also <http://libvirt.org/>.

³See also <https://www.docker.io/>.

inherent file system type on systems that have cgroups 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 on 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 subsystems⁴. Figure 3.1/24 illustrates a couple example hierarchies.

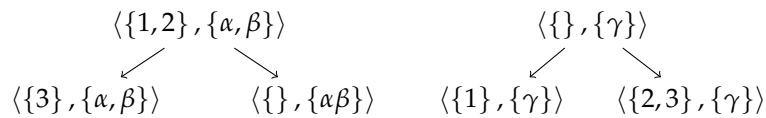


Figure 3.1: An illustration of two example cgroup hierarchies on a particular system. The set of identifiers of the tasks running on the system is $T = \{1, 2, 3\}$. The set of subsystems available on the system is $S = \{\alpha, \beta, \gamma, \delta\}$ (not all subsystems need be associated with a hierarchy). Every node (cgroup) has type $\mathcal{P}(T) \times \mathcal{P}(S)$, where \mathcal{P} denotes the powerset.

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 options⁵:

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

This associates the “cpu” and “cpuacct” subsystems with a new hierarchy, and mounts the hierarchy 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 on the system, this effectively means that a subsystem may be associated with at most one hierarchy. If there already exists a hierarchy, associated with the exact same set of subsystems, the hierarchy will be mirrored under the new target.

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, respectively. We monitor/modify the preferences of a cgroup by monitoring/modifying the control

⁴ [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 we have already discussed.

⁵Omitting the subsystem list, attempts to associate all subsystems available on the system.

files. We create/remove child groups by creating/removing subdirectories below our target.

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

`cgroup.procs`

Lists the set of thread group IDs in the current cgroup. Appending a thread group ID to this file attaches the thread group to this cgroup.

Hierarchical accounting

Hierarchical accounting when resource accounting is aware of the child groups of a cgroup. For instance, all resource usage is summed up for all tasks in the cgroup, and recursively for all child groups. Limits are then imposed on the subhierarchy rooted at the cgroup. A subsystem does not necessarily perform hierarchical accounting.

3.2.2 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.

3.2.3 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.

The memory subsystem uses a resource counter for a couple different memory resources. The resource counter control files (see also § [3.2.2/25](#)) 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.

3.2.4 `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.

3.2.5 `cpu`

The `cpu` subsystem facilitates CPU scheduling parameters for a cgroup.

The parameters facilitate control over e.g. the Completely Fair Scheduler (CFS) in the Linux kernel [[sched-design-CFS.txt](#)]. CFS is a proportional share

CPU scheduler. The CPU time is divided fairly among tasks depending on their priority and the share assigned to their cgroup.

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 each 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”, it has also become relevant to enforce upper bounds on the CPU time over a period of time [Turner et al. (2010)]. 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. The control file in the root cgroup, provides for a yard-stick for all other cgroups.

The `cpu` subsystem parameters 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 time of our programs.

3.2.6 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. Using `a`, 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 identifiers is what Linux uses to uniquely identify devices.

The following control files are of interest to us:

`devices.allow`

Writing an entry to this file adds an entry to the device whitelist.

`devices.deny`

Writing an entry to this file removes an entry from the device whitelist.

3.3 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 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.1/55), 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 directory 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.

3.3.1 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.

3.3.2 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.

3.3.3 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 processes within one namespace cannot communicate with processes in another namespace using these primitives.

3.3.4 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.

3.3.5 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 dropped for a new child namespace.

3.3.6 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 privilege escalation within the container, this would merely correspond to some unprivileged user on the host.

This opens up containers to a wide range of capabilities, 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 deserve to be tested further before being enabled by default [[Kerrisk \(2013\)](#), [Arch Linux Bug 36969](#), [Fedora Bug 917708](#)].

3.4 Resource Limits

The system call [[getrlimit\(2\)](#)], and its 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.

3.4.1 nfile

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.

3.4.2 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.

3.5 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.

3.5.1 SELinux

3.5.2 AppArmor

3.5.3 Capabilities

3.6 Seccomp

Chapter 4

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 be too obscure or too detailed for a student to benefit from it (see also § 1.1.2/9). 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 Assessment

Chapter 5

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 5.1/33. 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.

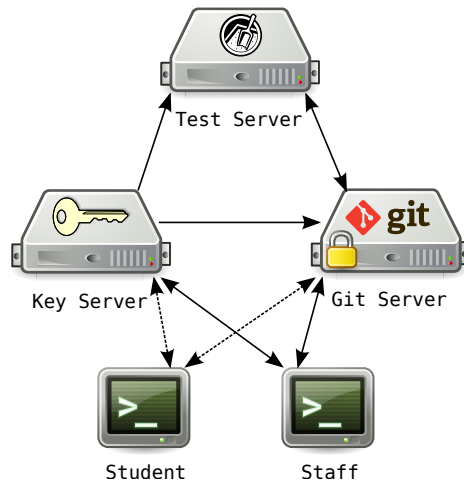


Figure 5.1: Illustration of some of the system architecture. All communication happens over secure channels. Both students and staff interact with the key server and Git server, but students have considerably less permissive rights on each of the servers

5.1 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 ideological 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.

5.1.1 Public Key Cryptography

5.1.2 OpenSSL

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

5.1.3 OpenPGP

5.2 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 § 5.3.8/41).

5.2.1 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.

5.2.2 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..

5.2.3 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..

5.2.4 Discussion

To be continued..

5.3 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 § 5.3.8/42 we will discuss why we have just one Git server.

5.3.1 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, on-line general purpose data store and gateway, having fine-grained and reliable authentication and authorisation procedures.

5.3.2 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.

5.3.3 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.

5.3.4 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.

5.3.5 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.

5.3.6 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](http://gitolite.google.com) (2011)].

There are also other tools out there, such as Gerrit¹ and Stash². 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](https://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.

¹See also <https://code.google.com/p/gerrit/>.

²See also <https://www.atlassian.com/software/stash>.

³See 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.

5.3.7 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 push⁴. This means that students cannot e.g. amend 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 [[Gerwitz \(2013\)](#)], 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.

5.3.8 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

⁴As 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.

5.4 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..

5.5 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/6Nwr8h817>.

[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-potentialerogrisicimedomslog.pdf> on March 9, 2014.

Archived by WebCite[®] at <http://www.webcitation.org/6NwjYw1Bi>.

[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/6NyIODlqJ>.

[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/6NzV0dWtv>.

[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/6070hF8LW>.

[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 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-Hill, 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/60GkhvGyE>.
- [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/60IAQzo2d>.
- [Valenti et al. (2003)] Salvatore Valenti, Francesca Neri, and Alessandro Cucchiarrelli. *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. Retrieved 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/60mayfnSi>.
- [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/60mbdAIQF>.
- [gitolite.com (2014a)] *Hosting git repositories*. Retrieved from <http://gitolite.com/gitolite/index.html> on April 13, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/60nsR9f7R>.
- [gitolite.com (2014b)] *authentication and authorization in gitolite*. Retrieved from <http://gitolite.com/gitolite/how.html> on April 13, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/60oAvdLeK>.
- [gitolite.com (2014c)] *who uses gitolite*. Retrieved from <http://gitolite.com/gitolite/who.html> on April 13, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/60oB1QVpA>.
- [gitolite.com (2014d)] *adding and removing users*. Retrieved from <http://gitolite.com/gitolite/users.html> on April 18, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/60vHM2Z2h>.

- [gitolite.com (2014e)] *different types of write operations*. Retrieved from <http://gitolite.com/gitolite/write-types.html> on April 18, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/60vIKAXXm>.
- [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/60oBgEPwG>.
- [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/60oCHMGop>.
- [openssh.com (2014)] OpenBSD. OpenSSH. Retrieved from <http://www.openssh.com/> on April 16, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/60sQXs51M>.
- [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/60yQjYQnz>.
- [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/60sTPwEgi>.
- [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/60sTL04Ce>.
- [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 April 16, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/60sZP6SgC>.
- [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 April 16, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/60t2V2qp1>.

- [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 April 17, 2014. Archived by WebCite[®] at <http://www.webcitation.org/60t6LflKS>.
- [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 April 17, 2014. Archived by WebCite[®] at <http://www.webcitation.org/60tvjunoB>.
- [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 on 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>.
- [jail(8)] FreeBSD 9.2. *JAIL(8). jail - manage system jails*. FreeBSD System Manager's Manual. Published on October 12, 2013. Retrieved from <http://www.freebsd.org/cgi/man.cgi?query=jail&sektion=8> on May 19, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/6PgDGfWV3>.
- [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/60t2yqQYR>.
- [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/60t4dU85N>.
Revision history retrieved from <http://stackoverflow.com/posts/432518/revisions> on April 17, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/60t4hGKpY>.
- [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/60t4qY1Y2>.
- [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/60vKd3HTW>.

- [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/60vM6eL7m>.
- [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/60vM1Tlsh>.
- [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/60vMGTWjE>.
- [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/60vK7dQBv>.
- [Gerwitz (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/60vKR8Hzi>.
- [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/60vLP3zYf>.
- [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/60vKurm25>.
- [Stefan (2013)] Deian Stefan. *cjail - a sandbox utility for Arch Linux*. Stanford Computer Security Lab. Last updated April 13, 2013. Source code retrieved from [git@github.com:scslab/cjail.git](https://github.com/scslab/cjail) on April 14, 2014.
- [DOMJudge (2014)] DOMJudge team. *DOMjudge Administrator's Manual*. Last published on May 9 2014. Retrieved from <http://www.domjudge.org/docs/admin-manual.pdf> on May 19, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/6PgRY124e>.

- [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/1/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/60xFtc1KV>.
- [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, Ontario, 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>.
- [Kerrisk (2012)] Michael Kerrisk. *LCE: The failure of operating systems and how we can fix it*. November 14, 2012. LWN.net. Retrieved from <http://lwn.net/Articles/524952/> on May 19, 2014.
Archived by WebCite[®] at <http://www.webcitation.org/6Pgfc67y>.
- [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/6PWMQA0oI>.
- [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.
- [Malan (2010a)] David J. Malan. 2010. *Reinventing CS50*. 2010. In Proceedings of the 41st ACM technical symposium on Computer science education (SIGCSE '10). ACM, New York, NY, USA, pp. 152–156.
- [Malan (2010b)] David J. Malan. 2010. *Moving CS50 into the cloud*. June 2010. J. Comput. Small Coll. Vol. 25, No. 6, pp. 111–120.
- [Malam (2013)] David J. Malan. 2013. *CS50 sandbox: secure execution of untrusted code*. 2013. In Proceeding of the 44th ACM technical symposium on Computer science education (SIGCSE '13). ACM, New York, NY, USA, pp. 141–146.
- [Graber (2014)] Stéphane Graber. *LXC 1.0 released*. Posted on February 21, 2014. Retrieved from <http://lwn.net/Articles/587545/> on May 19, 2014.
- Archived by WebCite® at <http://www.webcitation.org/6PgzTGt4q>.
- [LXC (v1.0.3)] LXC 1.0.3. Tagged by Stéphane Graber on April 9, 2014. Retrieved from <https://github.com/lxc/lxc/archive/lxc-1.0.3.tar.gz> on May 19, 2014.
- Archived by WebCite® at <http://www.webcitation.org/6PhsWna61>.
- [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)

iQIcBAABAgAGBQJTXE0zAAoJEDjbvchgkkmk+0AUQAKrfkqerXpwePAEHFCBqTqvN
R2fZa6tAY1w5psN8grWh2seu2W1KAteK53oht/6uZITqs3i2pYmRAJyEzVTBggs9
4BI0rClqebei03wkD1biRAIMPQWt6UAB/pvjBeMmMiw4G7FFZHSvBSct91Dsnb87
4A7083ZT/A421C20tH3vR0ehyQDyfHp+oL22SKMCoXKCCMCDZp5K07AMVggrzDoZ
```

```
KGDEeBpowCSctoUEEB1rVGz/syyaWzzcMy+UYeZ12JxpfgnX5oq14w1HIPfAhJn
/P6x70vmN75oIrxrt4rRs+aUY97iuiEzPpn9F2K4rNruTZUXN7906h/WWCJ/K/b0
D80wC1msaJqMYIEhQICu5kwezVswKVHz3QM9B01ak3Rg0bw3j70KKVxJQ95I6jYn
I3uz8RDGXWvp+6aso8v1/HWbQ6dCCA/9p1YALJZmRcy2Yg0A0nH3w6+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, Christoph 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 call¹. 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.

¹The 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.

Appendix B

Technical Details

This appendix covers some of the technical details related to this work.

B.1 Control Groups

This section covers some of the technical details of our use of cgroups. This is not a general introduction to cgroups. For that, we refer you back to § 3.2/23.

Many modern Linux distributions come with cgroups and many standard subsystems (as discussed in § 3.2/23) enabled. Your 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.

B.1.1 Managing cgroups

Mounting

A subsystem may be associated with at most one hierarchy. 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)].

B.1.2 memory

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

B.1.3 cpu

The `cpu` subsystem also facilitates control over the Real-Time scheduler (RT) of the Linux kernel [[sched-rt-group.txt](#)]. RT schedules real-time tasks, i.e. tasks for which it is important to meet deadlines. 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 do not want to grant students anyhow.

B.1.4 cpuset

The `cpuset` 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. Having few processor cores, and few memory nodes, this subsystem is of little use to us.

B.1.5 devices

The device node type, its major and minor identifiers of can be found using `stat`. For instance,

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

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

Reading `devices.allow` and `devices.deny` control files is prohibited. Instead, the current device whitelist for a cgroup can be inspected by reading the `devices.list` control file.

B.1.6 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.