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

Chapter 0

Preface

The intent of this work is to explore the use of computers in assisting in the assessment of practical work in computer programming courses.

0.1 Audience

This work is primarily aimed at computer programming educators. At least two other classes of readers should also feel at home in distinct parts of the report: The first few chapters deal with assessment in general, and how computers can assist in assessment. The remainder of the report deals with sandboxing capabilities in the Linux kernel, and using cryptography and Git to provide for a safe and secure course content management system.

The reader should be familiar with the basics of computer programming and a Unix-like operating system. Ideally, the reader should be familiar with the basics of C, Posix shell scripting, and Git.

0.2 Contributions

The contributions of this work, aside from proposing a framework for assessing student work with the assistance of computers, are in the technical aspects:

(a) Securely share a file system between a Linux container and its host.

(b) Encrypted Git repositories, allowing multiple collaborators.

0.3 References

We have preferred material published in relevant scientific journals, yet accessible to the general public. This can be complicated when dealing with such a community driven effort as Linux. Of course, Linux source code is meticulously reviewed, but in a matter which can at times bes regarded as inferior, and at times, superior to scientific review.

To ensure long-lasting access to publicly available, web-based resources, we have additionally archived most of them using WebCite®.

0.4 About the Author

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

When referring to "practice" wrt. teaching, 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. I have worked as a TA for DIKU under the following courses (for some, multiple times): Introduction to Programming, Object-oriented Programming and Design, Introduction to Algorithms, Compilers, Operating-systems and Multiprogramming, Statistical Methods for Machine Learning, and Advanced Programming.

I have also been extensively involved with a recent redefinition and redesign of the Object-oriented Programming and Design course at DIKU. I have been a student at DIKU since 2009.

0.5 Attachments

Source code is attached as the archive `src.tar.gz`.
MD5 sum: `96dd2b157c96c485c5961ee0fb76616e`.

0.6 Remark

We do not implement everything we seek out to complete in our analysis. Consider having a look at § 6/47 after reading § 2/16 to get a grasp of where the project stands.

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 learning[1] [Ramsden (1992), Pishghadam et al. (2014)]. This is done with the intent of providing feedback or certification, performing selection or comparison, improving learning processes, etc. [Bradfoot & Black (2004)].

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

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

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

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

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

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

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

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

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 courage student learning, and sometimes also at the end, to facilitate and encourage future learning.

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

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 compute 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-assissted 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 instanc, 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 Standford 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 § 5/38.

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 language[1]. 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 general[2].

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 interactive[3]. For a particular assignment, we make

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

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

[3] For 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 lockstep 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 loose 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 interface[1].

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-lxc[2] and Docker[3]. 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.1/63), and their children [cgroups.txt]. Some more technical details, as well as less related aspects of cgroups are discussed in Appendix B.1/64.

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

[1] Ideally, 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.

[2] See also `http://libvirt.org/`.

[3] See 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 subsystems[4]. Figure 3.1/24 illustrates a couple example hierarchies.

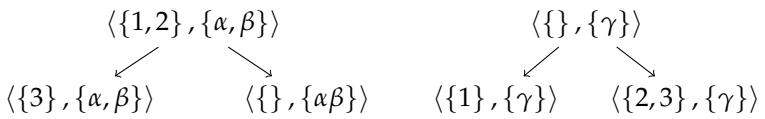

Figure 3.1: An illustration of two example cgroup hierarchies on a particular system. The set of identifiers of the tasks running on the system is $T = \{1,2,3\}$. The set of subsystems available on the system is $S = \{\alpha,\beta,\gamma,\delta\}$ (not all subsystems need be associated with a hierarchy). 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 options[5]:

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

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

A subsystem is busy e.g. if it is associated with a hierarchy having tasks attached. Since a hierarchy is automatically attached to all tasks on the system, this effectively means that a subsystem may be associated with at most one hierarchy. If there already exists a hierarchy, assosciated 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, respecitvely. 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.

[5] Omitting 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 proprtional 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 paramenters 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 (a)]. 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 thisfile 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 independent instance of the resource. Various namespace types abstract over various system resources. For each namespace type, a task is associated with exactly one namespace of that type.

Namespaces are hierarchical in the following a sense: A system boots with one global namespace for each namespace type. Tasks inherit their parent's namespaces by default. A task can be disassociated from a namespace and associated with another at runtime.

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, associates the process with an already existing namespace. The last is a general system call for task creation, allowing to create a task, already in a new namespace.

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, i.e. all of its ancestors. Typically, we'll only talk about a direct child-parent relationship.

We discuss various namespace types in the following subsections. Some types, such as mnt, pid, and net, require for a user to be priveleged to create a child namespace. We also discuss some technical details in Appendix B.2/66.

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 the new root within the container.

This allows us to subsequently unmount the old root, provided that the old rot is not busy. The old root is busy if there are mounts to targets under the old root, the running process originates, or has files open under the old root. By closing all open files, switching to a process originating from under the new root, and unmounting all mounts under the old root, we can get to unmount the old root.

[pivot_root(2)] 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 has to be properly remounted first. By mounting the new root in a

`tmpfs`, or perhaps even `squashfs`, we can also deny access to all block devices using the devices control group subsystem (see also § 3.2.6/27). This makes remounting the old root all the more cumbersome.

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. 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.

For simplicity, we'll 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. A new namespace has its own set of identifiers, starting at 0.

Hierarchies are implemented such that a user id in a container is mapped to a user id on the host (and likewise for groups). For instance, we can have a particular designated "container user" on the host, and map this user to UID 0 inside the container. This way, even if a malicious user managed to perform a privilege escalation within the container, this would merely correspond to an unprivileged user on the host.

This opens up containers to a wide range of capabilties, which would have otherwise required a privileged user on the host. For instance, containers

can now be created using an unprivileged user in general, by creating a user namespace first.

Arguably, this leaves too much of the kernel wide open for a container, and many find that user namespaces 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 siblings, `setrlimit(2)` and `prlimit(2)`, can be used to manage per-user soft and hard limits on various resources. In general, a process is warned upon reaching a soft limit, and killed, or prohibited in acquiring more of that resource, upon reaching a hard limit.

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. Limits are enforced per user session, and are enforced throughout the lifetime of a user session.

We discuss some interesting possible limits below. Some further technical details are discussed in Appendix B.3/66.

`RLIMIT_NPROC`
: A limit on the number of tasks a user can create per session.

`RLIMIT_CPU`
: A limit on the total CPU time per process.

`RLIMIT_CORE`
: A limit on the size of the core file.

`RLIMIT_NPROC`, combined with `RLIMIT_CPU` and a limit on the wall-clock time, can be used to mitigate for fork bombs, busy, and long-running processes. A fork bomb is a process that recursively creates new tasks in attempt to cause a denial-of-service. A limit on the wall-clock time ensures to kill off long-running processes which consume little to no CPU time, e.g. interactive processes waiting unduly for user input.

Putting a limit on the core file size can be a good idea to prevent students from inspecting us inspecting the runtime of their programs. This may reveal certain aspects of the system, which they in turn may use to exploit it.

There are other interesting limits, but combined with control groups, we have found these to be of little use.

3.5 Linux Security Modules

Linux Security Modules (LSM) is an access control framework. It provides a mechanism for various security checks to be raised whenever a kernel operation may result in an access control violation [Wright et al. (2002), LSM.txt].

There are a number of competing kernel extensions which hook into the LSM framework to provide for comprehensive security policies. We discuss some of these below.

3.5.1 SELinux

The Security Enhanced Linux® (SELinux) project predates LSM. It was initially envisioned inside the US National Security Agency (NSA). After 10 years of development and testing, it was released to the general public under a GPL license in 2000 [Ivashko (2012)].

When suggested to include it in the mainstream Linux kernel, Linus Torvalds charged the security community to make it a module instead, as there were other competing projects out there, e.g. AppArmor. The result was the implementation of LSM in the kernel [Wright et al. (2002)].

3.5.2 AppArmor

AppArmor is perhaps a more community-driven alternative to SELinux, which is presumably easier to use and more portable than SELinux [Spenneberg (2006)]. This sometimes comes at the cost of some security. For instance, with AppArmor, file system policies are enforced based on file paths, rather than inodes, protecting paths, not data.

3.5.3 Capabilities

Linux capabilities is an effort to subdivide the privileges of a privileged user into units. This allows to grant a user particular privileges, without making them privileged in general.

Linux capabilities is the default LSM for the Linux kernel [LSM.txt]. Rather intentionally, Linux capabilities provide only very coarse-grained access control options, compared to e.g. SELinux or AppArmor. For simplicity, we choose to deal with Linux capabilities rather than any other LSM.

Linux capabilities are useful to us in connection with user namespaces. We may designate a container user or group on the host, having extended capabilities. This user may be used to create a user namespace, creating the illusion full capabilities, but having only limited capabilities on the host.

3.6 Seccomp

Student programs may often be expected to use but a handful of system calls. Whitelisting the syscalls that students may use provides both for making sure that they use the intended tools, and that student programs cannot abuse obscure, or even vulnerable parts of the Linux kernel.

The secure computing (seccomp) part of the Linux kernel is provides for such syscall filtering [seccomp_filter.txt].

Seccomp was originally conceived by Andrea Arcangeli [Arcangeli (2005)] while working on the cpushare project, a service that would let you to contribute your idle CPU, for those in need of CPU time [LWN.net (2005)]. This requires executing arbitrary code on your computer, and so limiting the allowed syscalls to read/write/exit/sigreturn was a simple sandboxing idea. The Chromium team has since worked on extending this functionality [Tinnes (2012)] with an approach inspired by Berkley Packet Filters [Drewry (2012)].

Seccomp filters can be applied using the [prctl(2)] system call. We can use this to specify a filter program, that given a particular syscall will reply whether the syscall should be executed or not. Some sample filter programs can be found in [Linux kernel (v3.14.2)], under `./samples/seccomp/`.

3.7 Timeout

[timeout(1)] is a simple user space utility that allows to send a signal once a program exceeds a wall-clock time limit, e.g. `SIGTERM` or `SIGKILL`. The applicability of this has already been discussed in § 3.4/30.

Unfortunately, as we had realised too late, using [timeout(1)] skews CPU time measurements. The process and its children are frequently interrupted to check if they haven't exceeded the wall-clock time limit (see also § 6/47).

Chapter 4

Assessment Engine

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

— Doug McIlroy (1994)

Some of the more obscure uses of `pivot_root()` may quickly lead to insanity.

— `pivot_root(2)` (2012)

When a student makes a submission, it needs to be assessed. The assessment is specified by the teaching staff, and may consist of a range of static and dynamic analyses, as well as a feedback processing step.

In general, we refer to these steps as the *processes* of an assessment. The processes of an assessment, each run their own sandbox and communicate with other processes using standard in/out, and perhaps an in-memory file system. The combination of these sandboxed processes forms an *assessment engine*, taking the student submission in at one end, and producing "student-friendly" feedback at the other.

The general approach to designing these processes is intended to be to monitor a reference solution: which files, libraries does it need, how much memory and CPU time does it need, etc. This data is then used to create a conservative sandbox for running the processes of an assessment.

4.1 Sandboxes and Chaining

We follow the Unix philosophy [Salus (1994)] in our sandboxing approach, i.e. we write programs that do one thing and (eventually) do it well. More sophisticated sandboxes are achieved by chaining together simple sandboxing programs. A simple sandboxing program puts itself into a simple sandbox and executes (or forks and executes) the next program on its argument list.

This keeps the individual sandboxing quirks to the individual programs, and allows for an easy composition of sandboxes into sophisticated sandboxed environments. Unlike LXC, and other tools we've seen, it also gives us full control of the order in which a sandbox is constructed.

Some of our simple sandboxing programs are summarised below:

`cgroups`
: Attach the process to a set of cgroups. The cgroup `tasks` files (where to append the pid) are passed as options to the program.

`mnt`
: Enter a new mount namespace.

`user`
: Enter a new user namespace. The current user is mapped to uid 0 within the container.

`pumin`
: Enter a new (p)id, (u)ts, (m)ount, (i)pc, and (n)et namespace.

`sh-rootfs-bind`
: Bind-mount a root file system. By default, the subfolder `rootfs` is assumed to contain the new root file system, and `rootfs-target` is assumed to be the target for the mount.

`sh-rootfs-squashfs`
: Similar to `sh-rootfs-bind`, but mounts a root file system as a SquashFS. By default, assumes that the SquashFS file is located at `squashfs`.

`sh-iofs`
: Mounts a tmpfs for use for file-system-based input to and output from a container. By default, assumes that the input should be copied from `input`, and the output should be copied to `output` (after deleting all other files). Also, assumes that the tmpfs target is `tmpfs-target`, and under the new root, `rootfs-target/home/student`. Should only be called after the new rootfs has been mounted.

`pivot-root`
: Pivot the root file system. By default, it is assumed that the new root is under `root`, and the old root can be placed under `rootfs/.oldroot`.

`umount-oldroot`
: Unmount the old root file system. By default, assumed to be under `/.oldroot`.

`setuid`
: Set the user id. By default, to 1000.

`rlimits`
: Set resource limits. By default, set core file size to 0, the maximum number of processes to 20, and the maximum CPU time to 1 second.

We can chain these together to form sophisticated sandboxes. For instance, let `rootfs` be a subfolder in the current working directory, containing an executable `rootfs/home/student/forkbomb` (more on the necessity of `sudo` later):

```
$ sudo ./pumin ./sh-rootfs-bind ./pivot-root /home/student/forkbomb
```

The keen reader might notice that e.g. `setuid` and `pivot-root` already have Linux user space equivalents. We reimplement these to fit our chaining approach. We leave it to future work to not reimplement existing simple sandboxing programs, or patch these programs to suit our needs.

Remark: Under the current implementation, the default is typically the only option for the above programs.

4.2 Monitoring and Limits

In general, we would like to monitor the execution of our programs. For instance, we may monitor a reference solution to an assignment, so as to arrive at sound limits on submissions for that assignment. Monitoring a submission may also give some useful information wrt. feedback.

We introduce the program `monitor` performing monitoring and imposing limits. The program uses a sophisticated sandbox (see above) to impose many general limits, and allows optionally for a limit wall-clock time, CPU time, and memory. The program also has the option of using either a bind-mounted or a SquashFS root file system.

```
$ ./monitor -h
Usage: sudo ./monitor
  [-w <timeout>]
  [-m <memlimit-in-bytes>]
  [-t <cpu-time-limit-in-seconds>]
  [-s] (use squashfs for the rootfs)
  <executable-on-rootfs>
```

For instance, let's try monitoring a forkbomb that prints a `.` for every process it gets to spawn:

```
$ sudo ./monitor /home/student/forkbomb
..............Wall-clock time (seconds): 0.04
Max memory use (bytes): 524288
Total CPU time (nanoseconds): 3493968
Total CPU time in user mode (USER_HZ): 0
Total CPU time in kernel mode (USER_HZ): 0
```

These results come from putting the program into a memory, and cpuacct namespace, as well as making the [time(1)] Linux user space program part of our program chain.

You can experiment with this on the referenced virtual machine (see also § 6/47), or browse the code under the attached `./src/jail/monitor`. The results of applying various optional limits are also discussed in § 6/47. In the following sections we discuss some of the more interesting aspects of our sandboxes and monitoring.

4.3 Pivot Root

One particularly useful application of mount namespaces (see also § 3.3.1/28) 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 the new root, within the container.

This allows us to subsequently unmount the old root, provided that the old root is not busy. So [pivot_root(2)] can be used to hide 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 has to be properly remounted first. By mounting the new root in a `tmpfs`, or perhaps even `squashfs`, we can subsequently deny access to all block devices within the container (see also § 3.2.6/27). This makes remounting the old root all the more cumbersome.

4.3.1 Unmount vs. Detach Old Root

The old root is busy if there are mounts to targets under the old root, or the running process depends on the old root. By closing all open files, switching to a process originating from under the new root, and unmounting all mounts under the old root, we can get to unmount the old root.

However, unmount requires root privileges on the host. The approach above therefore requires that we pivot the root while still having root privileges on the host. We have not solved the problem of entering a user namespace after pivoting the root to some bare bones root file system. This is bad because it essentially renders user namespaces inapplicable in our case.

The other approach is to detach everything under the old root, meaning that it will be unmounted when the old root is no longer busy. We can then take a leap of faith that the old root will indeed be detached when we switch to a process under the new root (rendering the old root not busy). We have not had time to implement this.

4.4 Root File System

We prefer a bare bones root file system providing little more than exactly what is needed to execute our programs. It can be a bit challenging to assemble such root file systems.

We assemble a root file system by copying everything that we need under e.g. a `rootfs` subdirectory. For instance, in the attached source code, and for the referenced virtual machine (see also § 6/47), we have assembled:

`rootfs/lib64/ld-linux-x86-64.so.2`
: A dynamic linker for 64-bit executables on e.g. Ubuntu.

`rootfs/lib/x86_64-linux-gnu/libc.so.6`
: `glibc` for dynamic linking on e.g. Ubuntu.

`rootfs/.oldroot`
: Directory for the old root.

`rootfs/home/student`
: Directory for I/O file system.

With this root file system, we can execute any standard dynamically-linked 64-bit ELF executable built on our system.

We can figure out which dynamic linker a dynamically-linked ELF executable needs using `readelf(1)`[1]. Similarly, we can figure out some of the shared libraries that the executable needs[2]. However, once we've included the necessary dynamic linker, it will report if a shared library is missing. This way, we can assemble the necessary shared libraries using a fixed-point iteration.

Our programs may also use shell scripts, or require other obscure elements of the file system. To this end, we can use `strace(1)` (with the `-f` option) on our program to figure out what sort of files it is looking for and failing to find.

4.4.1 SquashFS

SquashFS is a compressed read-only file system for Linux[3]. In principle, it can serve as a file system type for read-only root file systems for our containers. Being read-only by design, this may provide some extra confidence that the root file system will remain read-only.

SquashFS does come with a couple quirks. A SquashFS is a file which can be mounted as a loopback device. The Linux kernel puts a bound on the number of loopback devices. This bound can be increased. Also, being mounted as a loopback device means that we can't blindly deny access to all devices. Luckily, loopback devices (and virtual character devices) always have the major device number 7 [devices.txt (b)].

4.5 I/O File System

One way to share data between the host and a Linux container is using a shared in-memory file system.

Once we've mounted a root file system (but before we pivot the root), we can mount a `tmpfs` under the current (old) root, and bind mount it under what will be the new root. Entering a new mount namespace, and pivoting the root moves all dependencies on the old root to the new root, and so the bind mound becomes a `tmpfs` mount inside the container, after the root is pivoted.

Of course, nothing happens to the mount outside the mount namespace, and the seemingly two `tmpfs` mounts indeed point to the same temporary file system. This way, we can share a file system between a host and container without giving access to a block device.

(See also the attached `./src/jail/sh-iofs` and `./src/jail/monitor`.)

[1] Look for "Required interpreter".
[2] Look for "Shared library".
[3] See also `http://squashfs.sourceforge.net/`.

Chapter 5

Course Content Management

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)

The intent of this chapter is to propose a course content management system (CCMS), allowing for interactive, automated assessment of student submissions. We consider this because the current CCMS in use at DIKU does not seemingly provide for easy hooks for student submissions, or even accessing and processing course data on our own account.

Our approach in general, is to use the Git version control system for the basic mechanism of content distribution. Teaching staff (and the assessment engine) use Git to publish learning material, assignments, and feedback. Students use Git to make submissions, and ultimately, collaborate on submissions.

Student data is encrypted and signed by the students. Access to the student data is mandated through an additional access control layer. This way we can ensure that student submissions indeed come from the students, and even if our access control layer fails to deliver, student data stays protected.

5.1 Git Server

The Git server serves as a data store and gateway between students and staff. In the following sections, we refer to students and staff collectively simply as clients, connecting to our host, the Git server. In § 5.8.4/45 we will discuss why we can make due with just one Git server.

5.1.1 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)].

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

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

5.1.2 Course as a Repository

A Git server manages Git repositories. Let a course be represented by a single Git repository. Let a commit by a student, to this repository, be a submission. The assignment to which the submission is made, depends on the where in the repository a commit makes a change.

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 the 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, and optionally, reported to the student upon submission. A submission is a particular commit by the student to their 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.

5.2 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 Git Hooks

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

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.

Other Git hooks are discussed in subsequent sections.

5.4 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 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 have also chosen to go with this option. We name our user, `git`.

5.5 Gitolite

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

Gitolite is used in multiple communities with high-stakes projects, such as Fedora, KDE, Gentoo, and kernel.org [gitolite.com (2014c)]. Among the reasons for choosing gitolite, kernel.org lists [kernel.org (2014)] “well maintained and supported code base”, “responsive development”, “broad and diverse

install base", and "had undergone an external code review" [gitolite Google Group (2011)].

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

5.5.1 Administration

Administration of the gitolite happens through a special Git repository on the Git server. 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 induce the users of the Git server [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.

5.5.2 Permissions

User permissions in gitolite can whitelisted for an entire repository, a branch, tag, or even a 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.

5.6 Encrypted and Signed Git Repositories

Git comes with a built-in a method to digitally sign tags (versions) and commits. The intent is to allow us to ensure that particular tags or commits come from particular individuals [Gerwitzh (2013)].

We could use this to ensure that a particular submission indeed comes from a particular student, but we cannot expect the students to be proficient enough in Git to use signed tags or commits. Instead, we consider Git repositories where commits are always, seamlessly signed.

At the core of Git is a key-value store [Chacon (2009)]. All version-controlled data, is retained in so-called Git objects, addressed by the SHA-1 hash of their contents. We say that commit data to, and checkout data from this data store.

Git retains diffs of data (compared to its earlier versions), rather than the data itself. Data in a local Git repository is therefore saved twice, once on the

[1] See also `https://code.google.com/p/gerrit/`.
[2] See also `https://www.atlassian.com/software/stash`.

regular file system, and once as a sequence of diffs, retained as Git objects. When performing a Git push operation to a remote Git repository, only Git objects are ever sent along.

The `filter` Git attribute [gitattributes(5)] allows us to filter the data before it is committed to, or when it is checked out of the data store. This is achieved by two commands: `smudge` and `clean`, respectively. The names suggest that the intended use filters is to retain only "clean" data in the data store.

We can abuse the system a bit, taking inspiration from its own documentation [gitattributes(5)]. We sign the data on commit and verify the signature on checkout. Similarly, we can encrypt the data that goes into the data store, and decrypt the data when it leaves the data store. This way, student data stays private, even if their personal Git objects leak from the Git server.

5.6.1 Keys

GNU Privacy Guard (GPG) is a suite of cryptographic tools. It supports, among other things, private/public key cryptography, and boasts tools for trustworthy distribution of public keys. GnuPG is often used to identify physical individuals on the Internet. Indeed, the built-in Git tag and commit signing features expect users to retain a GPG private/public key pair, and securely distribute their public keys to all interested parties.

OpenSSL is an open-source implementation of the SSL and TLS protocols, used for secure communication on the Internet. Unlike, GnuPG, OpenSSL is not intended for the identification of physical individuals. OpenSSL however, is often used to identify users on servers. For instance, OpenSSL is an underlies OpenSSH, which we used to identify users on our Git server (see also § 5.2/39).

Both GnuPG and OpenSSL can be used to implement encrypted and signed Git repositories. GnuPG however is more conventional for the identification of individuals, and it is compatible with OpenSSH [gpg-agent(1)]. OpenSSL is also very badly documented[3] and has recently suffered a severe drop in popularity due to a major vulnerability [cvedetails.com (2014c)].

Students can make due with securely informing us of their public GPG key, and use this to both interact with our Git server, encrypt, and sign their submissions.

We have not found a way to make public key distribution any more simple for students, than using some kind of a key server, where keys are validated with the participation from the teaching staff. One easier approach might be to make use of their online university accounts and have them supply their keys on their own in a way similar to Github. We have not had time to look into this.

5.6.2 Collaboration

It is easy enough to sign and encrypt the data for one user, but this defeats many of the useful collaboration features of Git. Besides, we must let the staff and the assessment engine see the student data as well.

[3] [openssl(1)] does not even mention that it can be used to encrypt, decrypt and sign data.

One way in which we can provide for collaborative, encrypted Git repositories is to encrypt the data for multiple recipients. This is straight-forward using the OpenPGP suite of tools [gpg2(1)]. For an implementation of this, see the attached `./src/git/git-init.sh`.

The pitfall of this approach in general is that new individuals cannot easily join as collaborators as the entire history has not been encrypted for them to see. This is not a problem if we do not let students collaborate on submissions, but again, this defeats many of the useful collaboration features of Git.

Another limiting factor to student collaboration is our choice of having students submit to individual branches. One way to let students collaborate, is to have them be allocated a group branch by the staff.

5.7 Attack Surface

In this section, we discuss some of the possible ways that an attacker can approach our Git server. This is distinct from attempting to attack our sandboxed environments in that the Git server is not sandboxed in any relevant way. It serves as a general purpose data store and gateway between students and staff.

5.7.1 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)], and disabling login for that user in general.

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.

5.7.2 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.

5.7.3 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 and staff are simply not allowed to perform a forced push, supporting the notion introduced in § 2.6/19, that course data is never deleted. 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.

5.7.4 Git, OpenSSH, and Perl

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

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

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

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

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

5.8 Discussion

5.8.1 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. Often, a developer would work on their own branch (as our students would), and then make a "pull request" to merge their changes into the master branch[4].

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.

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".

[4] Alternatively, a developer might work on in their own repository, and then make a pull request for their changes to be merged into the main project repository [Bird et al. (2009].

(This would demand a more complicated access control layer.) 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. In our model, the commit and assessment history of a branch is sufficient to reveal when the code had last worked.

5.8.2 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.

5.8.3 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 authorized 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.

5.8.4 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. 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.

Chapter 6

Discussion

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

We've sketched some aspects of a system that may serve as a framework for performing automated assessment of practical work in computer programming courses. We've discussed many technologies that we can use both to provide sandboxed execution environments, as well as a safe and secure course management system. The techniques discussed provide for lightweight system, capable of guaranteeing a high level of safety, security and fairness in performing assessments and managing course data.

We've also implemented some basic sandboxing capabilities that can be chained together to form more complicated sandboxes. Notably, we can communicate with our sandboxed programs using standard in, out, the file system, and even trace them with e.g. `strace(1)`.

Unfortunately, due to time constraints, the techniques described have only been tried out in a scratchbook fashion. They have not been all compiled into a system that is ready for any course at the time of writing. In particular, an infrastructure for static and dynamic analyses, as well as feedback processing is missing. Never-the-less we believe that this will be easy to assemble given the work done here.

6.1 Virtual Machine

For the purposes of some discussion, we've created an Open Virtualization Appliance (OVA) using VirtualBox, running Ubuntu 14.04, server edition. We chose Ubuntu, because it comes with user namespaces enabled by default (see also § 3.3.6/29). The OVA is intended to demonstrate some of the techniques discussed above.

We'll just be using one of the users on the system `onlineta` – the system administrator.

You should be able to log into the OVA using SSH, once booted on your host. For instance,

```
$ ssh -p 2222 onlineta@127.0.0.1
```

If not, you may need to set up port forwarding in your OVA client. In the case above, we forward host port 2222 to guest port 22.

Remark: For simplicity, we've disabled swap.

6.2 Monitoring and Limits

Under `/home/onlineta/jail` you will find the sandboxing programs discussed in § 4.1/33, as well as the `monitor` program discussed in § 4.2/35.

Although we've implemented a farily sophisticated sandbox, we have not made use of the following concepts:

- User namespaces.
- Capabilities.
- Seccomp.
- Disabling block device access.
- Detaching rather than unmounting the root file system (see also § 4.4/36).

Notably, the `monitor` program also has to be run with `sudo` as it needs to perform mounting operations — something that, to our knowledge, requires root privileges on the host.

6.2.1 The Observer Effect

Our `monitor` program provides for a couple options for monitoring and limits. We've noticed that applying a timeout, or using SquashFS has considerable effects on our monitoring results. We present these results below together with an execution of `monitor` using bind mounts and no timeout (we pressed a key after some time).

```
$ sudo ./monitor /home/student/stall
Press any key to stop stalling..
Wall-clock time (seconds): 8.29
Max memory use (bytes): 221184
Total CPU time (nanoseconds): 3019991
Total CPU time in user mode (USER_HZ): 0
Total CPU time in kernel mode (USER_HZ): 0
```

```
$ sudo ./monitor -s /home/student/stall
Press any key to stop stalling..
Wall-clock time (seconds): 4.50
Max memory use (bytes): 3100672
Total CPU time (nanoseconds): 13034532
Total CPU time in user mode (USER_HZ): 0
Total CPU time in kernel mode (USER_HZ): 1
```

```
sudo ./monitor -w 5 /home/student/stall
Press any key to stop stalling..
Wall-clock time (seconds): timeout
Max memory use (bytes): 180224
Total CPU time (nanoseconds): 4942463093
Total CPU time in user mode (USER_HZ): 161
Total CPU time in kernel mode (USER_HZ): 335
```

So using SquashFS here yields a 15x increase in memory use ise and 4x increase in CPU time, i.e. the lowest limit on memory we can set on memory is about 24 megabytes. Using a timeout there yields over 1600x increase in CPU time. This yields timeouts unusable together with limits on the CPU time.

This is unfortunate, as we would have liked more fine grained control of how much memory, CPU time, and wall-clock time students may use. At the same time, dropping a timeout limit allows student programs to stall. Using bind mounts ahead of SquashFS disallows from disabling access to block devices in general. We may be able to solve the latter problem using an approach similar to our I/O file system, but we have not had time to look into this.

Recall that we already showed that we safely catch fork bombs in § 4.2/35.

6.3 Future Work

We find that our approach to Linux containment is an interesting contrast to the rest. Combining simple, lightweight sandboxes into more sophisticated ones seems more true to the "Unix philosophy". It also gives us greater control of how our sandboxes are set up.

A lot of work remains to be done with our individual sandboxes, making them even more robust, independent, and lightweight. Also, a lot of sandboxing capabilities stand to be added, for instance, seccomp.

Clearly, we need to apply the techniques discussed above and combine them into a system usable for a course.

Once we do, we are likely to realise needs for a range of user space tools for the configuration of our assessment engines. For instance, tools that can automatically figure out which shared libraries should be included in a root file system for a particular dynamically linked program to be sandboxed.

If we ever move on to actually use such a system in a real course, we will probably observe a lack of general tools for the testing of student programs. This may become a large area of interest in and of itself — to discover a useful set of testing tools for all the conceivable courses.

6.3.1 Course Content and Key Management

Using Git for course content management may not be very user-friendly for students in disciplines other than computer programming. Even beginning computer programmers might find it perplexing.

Git gave us a simple way to retain multiple submissions to assignments, multiple versions of assignments, and multiple versions of assignment feedback. Git also gave us useful hooks which we used to provide for an encrypted

data store and a way for students to digitally sign their submissions. We also used Git hooks to automatically trigger the assessment of a submission when it arrives.

An interesting direction of future work would be to provide a more user-friendly interface to all this, or find a way to achieve similar objectives with e.g. a web-based platform. The major incumbent to this is our idea of students encrypting and signing their submissions. One source of inspiration, might be the new HTML5 `<keygen>` element[1].

[1] See also http://www.w3.org/html/wg/drafts/html/master/forms.html.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Chacon (2009)] Scott Chacon. *9.2 Git Internals - Git Objects*. Pro Git. 2009. Apress. Retrieved from http://git-scm.com/book/en/Git-Internals-Git-Objects on May 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6Pjx79c87.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[getrlimit(2)] Linux. *GETRLIMIT(2). getrlimit, setrlimit, prlimit - get/set resource limits*. Linux Programmer's Manual. Published 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.

[capabilities(7)] Linux. *CAPABILITIES(7). `capabilities` - overview of Linux capabilities*. Linux Programmer's Manual. Published on April 9, 2014. Retreived from http://man7.org/linux/man-pages/man7/capabilities.7.html on May 20, 2014.

Archived by WebCite® at http://www.webcitation.org/6Pi9UTD9J.

[prctl(2)] Linux. *PRCTL(2). `prctl` - operations on a process*. Linux Programmer's Manual. Published on April 14, 2014. Retrieved from http://man7.org/linux/man-pages/man2/prctl.2.html on May 20, 2014.

Archived by WebCite® at http://www.webcitation.org/6PiXs1kvs.

[gitattributes(5)] Git 1.9rc1. *GITATTRIBUTES(2). `gitattributes` - defining attributes per path*. Git Manual. Published on January 30, 2014. Retrieved from http://man7.org/linux/man-pages/man5/gitattributes.5.html on May 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6PjvQBK3t.

[timeout(1)] GNU coreutils 8.22. *TIMEOUT(1). `timeout` - run a command with a time limit*. User Commands. Published in May 2014. Retrieved from http://man7.org/linux/man-pages/man1/timeout.1.html on May 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6PkAc1NYL.

[time(1)] *TIME(1). `time` - time a simple command or give resource usage*. Linux User's Manual. Published on November 14, 2008. Retrieved from http://man7.org/linux/man-pages/man1/time.1.html on May 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6PkB4bYOP. i

[gpg-agent(1)] *GPG-AGENT(1). `gpg-agent` - Secret key management for GnuPG*. Retrieved from http://linux.die.net/man/1/gpg-agent on May 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6Pk2rDgJd.

[gpg2(1)] *GPG2(1). `gpg2` - OpenPGP encryption and signing tool*. Retrieved from http://linux.die.net/man/1/gpg2 on May 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6Pk8OU4Ce.

[openssl(1)] *OPENSSL(1). `openssl` - OpenSSL command line tool*. Retrieved from https://www.openssl.org/docs/apps/openssl.html on May 21, 2014.

Archived by WebCite® at http://www.webcitation.org/6Pk2dm4kX.

[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/6Ot2yqQYR.

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

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

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

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

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

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

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

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

[cvedetails.com (2014b)] CVE Details. The ultimate security vulnerability datasource. *Openssh: Security Vulnerabilities*. Retrieved from http://www.cvedetails.com/vulnerability-list/vendor_id-7161/Openssh.html, on April 18, 2014.

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

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

cvedetails.com/vulnerability-list/vendor_id-217/Openssl.html, on April 18, 2014.

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

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

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

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

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

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

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

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

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

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

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

[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 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/l/linux-2.6/news/20110601T223515Z.html on April 25, 2014.

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

[Wright et al. (2002)] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. *Linux Security Modules: General Security Support for the Linux Kernel*. August 2002. In Proceedings of the 11th

USENIX Security Symposium. USENIX Association. San Francisco, California, United States. Retrieved from https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright.pdf on April 19, 2014.

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

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

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

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

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

[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/6PgfCc67y.

[LWN.net (2005)] corbet. *Securely renting out your CPU with Linux*. Posted on January 26, 2005. LWN.net. Retrieved from http://lwn.net/Articles/120647/ on May 20, 2014.

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

[Arcangeli (2005)] Andrea Arcangeli. *seccomp for 2.6.11-rc1-bk8*. Email sent to Andrew Morton on January 21, 2005. Retrieved from http://lwn.net/Articles/120192/ on May 20, 2014.

Archived by WebCite® at http://www.webcitation.org/6PiW2e6nv.

[Drewry (2012)] Will Drewry. *dynamic seccomp policies (using BPF filters)*. Email sent to the Linux kernel mailing list on January 11, 2012. Retrieved from http://lwn.net/Articles/475019/ on May 20, 2014.

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

[Tinnes (2012)] Julien Tinnes. *A safer playground for your Linux and Chrome OS renderers*. November 19, 2012. The Chromium Blog. Retrieved from http://blog.chromium.org/2012/11/a-safer-playground-for-your-linux-and.html on May 20, 2014.

Archived by WebCite® at http://www.webcitation.org/6PiWcAvi8.

[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 20, 2014.

Archived by WebCite® at http://www.webcitation.org/6PhsWna6l.

[Ivashko (2012)] Evgeny Ivashko. *Secure Linux: Part 1. SELinux – history of its development, architecture and operating principles*. May 30, 2012. IBM DeveloperWorks. Retrieved from http://www.ibm.com/developerworks/linux/library/l-secure-linux-ru/ on May 20, 2014.

Archived by WebCite® at http://www.webcitation.org/6Pi7LvHOL.

[Spenneberg (2006)] Ralf Spenneberg. *Shutting out intruders with AppArmor — Protective Armor*. August 2006. Linux Magazine, No. 69. Retrieved from http://www.linux-magazine.com/content/download/63099/487070/version/1/file/Shutting_out_Intruders_with_AppArmor.pdf on May 20, 2014.

Archived by WebCite® at http://www.webcitation.org/6Pi8B88Eg.

[Salus (1994)] Peter H. Salus. *A Quarter-Century of Unix*. 1994. Addison-Wesley. ISBN 0-201-54777-5.

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

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

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

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

Yielding the signature:

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

iQIcBAABAgAGBQJTXE0zAAoJEDjbvchgkmk+0AUQAKrfkqeRXpwePAEHFCBqTqvN
R2fZa6tAY1w5psN8grWh2seu2W1KAtEk53oht/6uZITqs3i2pYmRAJyEzVTBggs9
4BI0rClqebeiO3wkD1biRAIMPQWt6UAB/pvjBeMmMiw4G7FFZHSvBSct91Dsnb87
4A7083ZT/A421C2OtH3vROehyQDyfHp+oL22SKMCoXKCCMCDZp5K07AMVggrzDoZ
KGDEeBpowCSCtoUEEBlrVGz/syyaWZzzcMy+UYeZ12JxpfgnX5oq14w1HIPfAhJn
/P6x7OvmN75oIrxrt4rRs+aUY97iuiEzPpn9F2K4rNruTZUXN7906h/WWCJ/K/b0
D80wC1msaJqMYIEhQICu5kwezVswKVHz3QM9BO1ak3RgObw3j70KKVxJQ95I6jYn
I3uz8RDGXWvp+6aso8vl/HWbQ6dCCA/9plYALJZmRcy2Yg0A0nH3w6+ckC1x/r4l
ZyR6NEcVYg27HQswjmWxbqUhapFMLQGj5oGZ9svbsdwet3ckQTcqAtS5N/YHZZaQ
SnzvY4dZ/MoRwdCGz0hC99RofIgMPgY8ypkc2GGvyGv9uDLsK4koB65ZX1zW/oRw
43eatEoY/Q1QyGWrbwqEWFY91XbZne1KJNwdXYkmTDawMI2F2zApIjsAHpMseJiN
XZPAJqjjAF6nhxRzsrI8
=HZrL
-----END PGP SIGNATURE-----
```

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

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

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

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

[devices.txt (a)] 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/`.

[seccomp_filter.txt] Andy Lutomirski. *SECure COMPuting with filters*. Found in [Linux kernel (v3.14.2)], under `./Documentation/prctl/`.

[devices.txt (b)] Maintained by Alan Cox. *LINUX ALLOCATED DEVICES (2.6+ version)*. Last revised: 2009-04-06. Found in [Linux kernel (v3.14.2)], under `./Documentation/`.

Appendix A

General Linux Concepts

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

A.1 Tasks

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

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

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

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

Appendix B

Technical Details

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

B.1 Control Groups

This section covers some useful technical details 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 cgorups

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

We choose to let the system be runnable on commodity hardware. 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` to `devices.allow` or `devices.deny` is the same as writing `a *:* rwm` to the same file.

Reading `devies.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 block 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.

Note, this does not inhibit us in providing students with a general purpose read/write file system.

B.2 Namespaces

This section covers some useful technical details of namespaces. This is not a general introduction to namespaces. For that, we refer you back to § 3.3/28.

The namespaces that a task is associated with can be listed using the [proc(5)] pseudo file system. There is a pseudo file `/proc/[pid]/ns/[nstype]`, for each thread group ID (`[pid]`) on the system, and for each namespace type (`[nstype]`) enabled on the system.

You can check which namespace types 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 namespaces subdiretory of the init process:

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

Not all namespace types supported by the Linux kernel are enabled by default in mainstream Linux distributions. As discussed in § 3.3.6/29, the user namespace is frequently disabled by default due to security concerns. To our knowledge, enabling support for a namespace type requires running a custom kernel in general.

B.3 Resource Limits

This section covers some useful technical details of resource limits. This is not a general introduction to resource limits. For that, we refer you back to § 3.4/30.

Having a designated "container" user or group, we can enforce some of the resource limits at session set up using Pluggable Authentication Modules (PAM). In particular, the [pam_limits] module. This module however, does not offer as fine-grained limits as the kernel primitives. For instance, it only allows to limit CPU time in terms of minutes, not seconds, as the kernel primitives.

B.4 Installing Gitolite

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

The code is distributed under a GNU General Public License, and is available at git://github.com/sitaramc/gitolite. We may wish to check out the latest tag (version), after verifying that it indeed was signed by Sitaram Chamarty (the original developer of Gitolite)[1]. To the best of our knowledge, his public GPG key is:

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

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

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

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

[1] See also http://git-scm.com/book/en/Git-Basics-Tagging, if you are unfamiliar with Git's tagging mechanism.